Kachina - Foundations of Private Smart Contracts

KACHINA – Foundations of Private Smart Contracts

Thomas Kerber
The University of Edinburgh, IOHK
papers@tkerber.org

Aggelos Kiayias
The University of Edinburgh, IOHK
akiayias@ed.ac.uk

Markulf Kohlweiss
The University of Edinburgh, IOHK
mkohlwei@ed.ac.uk

***Abstract*—Smart contracts present a uniform approach for deploying distributed computation and have become a popular means to develop security critical applications. A major barrier to adoption for many applications is the 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 unified method 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 core protocol, KACHINA, for deploying privacy-preserving smart contracts, which encompasses previous systems. We demonstrate the KACHINA method of smart contract development, 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.

I. INTRODUCTION

Distributed ledgers put forth a new paradigm for deploying online services beyond 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 relevant business logic. Instead, services can take advantage of decentralized, "trustless" computation to improve their transparency and security as well as reduce the need for trusted third parties and intermediaries.

Bitcoin [23], the first successfully deployed distributed ledger protocol, does not lend itself easily to the implementation of arbitrary protocol logic that can support this paradigm. This led to many adaptations of the basic protocol for specific applications, such as NameCoin [16], a distributed domain registration protocol, or Bitmessage [28], a ledger-based communications protocol. An obvious problem with this approach is that, even though the Bitcoin source code can be copied arbitrarily often, 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 [26], were popularized by Ethereum [30], 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* [24]: 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 a distributed ledger.

A seemingly inherent limitation of the decentralized 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 [15], and secure-computation [14, 8]. Motivated by such cryptographic techniques, systems satisfying various definitions of privacy – and requiring various trust assumptions – have been proposed [2, 20, 31, 17], as we detail in Subsection I-B. Their reliance on trust assumptions nevertheless fundamentally limits the level of decentralization which they can achieve, especially compared to 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 decentralized 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, which we collectively refer to as "KACHINA core contracts". In particular, this includes contracts with privacy guarantees, which can be implemented without additional trust assumptions beyond what is assumed for Nakamoto consensus and the existence of a securely generated common reference string. 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, and can be reduced to the same assumptions as the distributed consensus algorithm itself [19]. This class allows us to express the protocol logic of dedicated privacy-preserving, ledger-based protocols such as Zerocash [1] as smart contracts. Existing smart contract systems such as Zexe [2], Hawk [20], Zether [4], Enigma [31], zkay [25], and Arbitrum [17] can be expressed, preserving their privacy guarantees, as KACHINA contracts. These protocols mainly rely on either *zero-knowledge* or *signature authentication* for their security. KACHINA is flexible enough to

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

A. Our Contributions

We make four contributions to the area of privacy-preserving smart contracts:

a) We **model** privacy-preserving smart contracts.
b) We **realize a large class** of such contracts.
c) We **enable concurrent interactions** with smart contracts, without compromizing on privacy.
d) We demonstrate a general methodology to **efficiently and composably build** smart contract systems.

Combined, they provide a method for both reasoning about privacy in smart contracts, and construct an expressive foundation to build smart contracts with good privacy guarantees upon.

a) Our model: We provide a universally composable model for smart contracts in the form of an ideal functionality that is parameterized to model contracts both with and without privacy, capturing a broad range of existing systems. 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 means to define their security, and contrast their security with other systems.

We consider a smart contract to be specified by a transition function Δ and a leakage function Λ, which parameterize the smart contract functionality $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$. Δ models the behavior of the contract, were it to be run locally or by a trusted party. 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 [7], and captures the fundamental ideal-world leakage through the parameterizing function Λ.

Some combinations of Δ and Λ are not obviously realizable, in particular the more restricted the leakage becomes. They are able to capture existing smart contract systems however, both privacy-preserving and otherwise. For instance, a leakage function which leaks the input itself corresponds closely to Ethereum [30], while a leakage function returning no leakage makes many transition functions hard or impossible to realize. This paper focuses on a more interesting middle ground. By defining the ideal behavior to interact with $\mathcal{G}_{\mathsf{ledger}}$, we avoid having to duplicate the complex adversarial influence of ledger protocols. We make few assumptions about this ledger, requiring only the common prefix property, and interfaces for submitting and reading transactions to be well defined.

b) Our protocol: We construct a practical protocol for realizing many privacy-preserving smart contracts, utilizing only non-interactive zero-knowledge. The primary goal of this protocol is to provide a sufficiently low-level and general purpose basis for further privacy-preserving systems, without requiring the underlying system to be upgraded with each new extension or change. We focus on the Nakamoto consensus setting of a shifting, untrusted set of parties. The protocol's core idea is to separate a smart contract's state into a *shared, on-chain, public* state, and an *individual, off-chain, private* state for each party. Parties then prove in zero-knowledge that they update the public state in a permissible way: That there exists a private state and input for which this update makes sense.

c) Dealing with concurrency in a privacy-preserving manner: There exists a fundamental conflict between concurrency and privacy that needs to be accounted for to remain true to our objective of providing a smart contract functionality as decentralized as Nakamoto consensus. To illustrate, suppose an ideal smart contract is at a shared private state ϕ and two parties wish to each apply a function f and g respectively to this state. They wish (in this specific case) the result to be independent of the order of application – i.e. $f(g(\phi)) = g(f(\phi)) = \phi'$. In any implementation of the above in which parties do not coordinate, the first party (resp. the second) should take into account the publicly known encoding $[\phi]$ of ϕ and facilitate its replacement with an encoded state $[f(\phi)]$ (resp. $[g(\phi)]$) as it results from the application of the desired transition in each case. It follows that the encoded states $[f(\phi)], [g(\phi)]$ must be publicly reconciled to a single encoded state $[\phi']$ 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 which operations are performed on the contract's state, when interacting with it through oracle queries. These allow contract authors to optimize when transactions are in conflict: ensuring minimal leakage occurs while still allowing reorderings. We provide a mechanism for analyzing when reordering transactions is safe with respect to a user's individual private state, by specifying a sufficient condition for when transactions must be declared as dependencies.

d) 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 ϕ was modeled anyway, (e.g. as inputs to these systems), the zero-knowledge proofs involved would scale poorly, with a proving complexity of $\Theta(|\phi|)$ before any computation is performed. A naive 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.

Regardless of the size of our state, the state is never accessed directly, but only through oracles specified by the contract. As a result the complexity of what must be proven is under the full control of the contract author, and can be optimized for. A proving complexity of $\Theta(|\mathcal{T}_\rho| + |\mathcal{T}_\sigma|)$ prior to performing any computation can be expected in KACHINA, where $\mathcal{T}_\rho$ is oracle transcript for the private state, and $\mathcal{T}_\sigma$ is the one for

the public state. This constitutes a clear improvement, as the state of smart contracts deployed in practice may be very large, however transcripts, similar to the inputs and outputs of traditional public contracts, are generally short. This increase in efficiency allows us to construct an entire smart contract system, akin to Ethereum [30], *as a* KACHINA *contract* in [18, Appendix J].

Not all contracts a user wishes to write will directly match the requirements for realizing a smart-contract with the KACHINA core protocol. However, our model is sufficiently flexible to allow direct application of the transitivity of UC-emulation to solve this: If the originally specified "objective" contract (Δ, Λ) is not in the class of KACHINA core contracts, the author can find an equivalent (Δ', Λ') which is. The author can provide a proof that $\mathcal{F}_{\text{sc}}^{\Delta',\Lambda'}$ UC-emulates $\mathcal{F}_{\text{sc}}^{\Delta,\Lambda}$, and by the transitivity of UC-emulation, can use the KACHINA core protocol to realize (Δ, Λ). We facilitate such proofs by including adversarial inputs and leakages in our model, which allow the simulator limited control over the objective smart contract. This method to develop private smart contracts is illustrated in Figure 1. It is further showcased by the implementation of the salient features of Zerocash [1] as a KACHINA contract in Section V, and the proof that it UC-emulates a much simpler ideal payments contract.

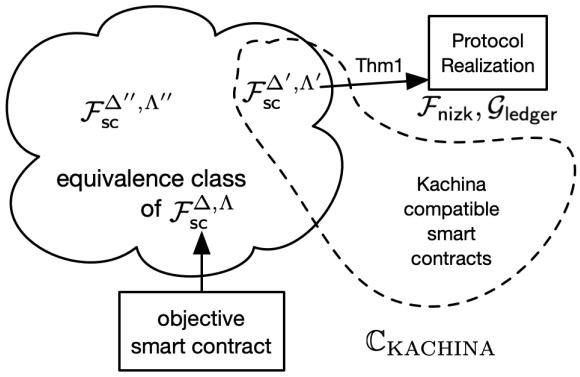

Fig. 1. An overview of the KACHINA method to develop private smart contracts: 1) An intuitive description of the objective smart contract is developed in the form of $\mathcal{F}_{\text{sc}}^{\Delta,\Lambda}$. 2) A KACHINA compatible $\mathcal{F}_{\text{sc}}^{\Delta',\Lambda'}$, from the set of all equivalent contracts $\mathcal{F}_{\text{sc}}^{\Delta'',\Lambda''}$ is selected, and the equivalence proven. 3) Theorem 1 is applied to obtain its realization.

B. 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 the most notable of these.

a) Ethereum: As the first practically deployed smart contract system, Ethereum [30] is the basis of a lot of our expectations and assumptions about smart contracts. Ethereum is not designed for privacy, and hides no data by itself. We assume that the reader is familiar with Ethereum.

b) Zexe: Zerocash [1] is a well-known privacy-preserving payment system, allowing direct private payments on a public ledger. Zexe [2] extends its expressiveness by allowing arbitrary scripts, reminiscent of Bitcoin-scripts, to be evaluated in zero-knowledge in order to spend coin outputs. It is a major improvement in expressiveness over Zerocash, which only permits a few types of transactions.

c) zkay: zkay [25] extends Ethereum smart-contracts with types for private data. It allows users to share encrypted data on-chain, and prove that data is correctly encrypted and correctly used in subsequent interactions. These proofs are managed through the ZoKrates [11] framework, which compiles Ethereum contracts into NIZK-friendly circuits. Its usage is limited to fixed size pieces of private data.

d) Hawk: One of the earliest works on privacy in smart contracts, Hawk [20] 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 they are corrupt, however they 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.

e) Zether: A lot of work on privacy in smart contracts has focused on retro-fitting privacy into existing systems. Zether [4], for instance, constructs a privacy-preserving currency within Ethereum, which can be utilized 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 generalize to many applications.

f) Enigma: There are two forms of Enigma: A paper discussing running secure multi-party computation for smart contracts [31], and a system of the same name designed to use Intel's SGX enclave to guarantee privacy [12]. 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 [3].

g) Arbitrum: Using a committee-based approach, Arbitrum [17] 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 a complexity logarithmic in 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.

h) State channels: State channels, such as those discussed in [10], 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.

i) Piperine: Piperine [21] uses a similar model and approach as presented in this paper, relying on zero-knowledge proofs of correct state transitions, and modeling smart contracts as replicated state machines. Piperine focuses on efficiency gains from this approach, rather than privacy gains, which it does not capture, while our work does not account for the benefit of transaction batching. Our notion of state oracles can be seen as a generalization of the state interactions presented in [21].

II. Technical Overview

We first informally establish 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 technical report [18]. 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.

a) Our model: We model smart contracts as *reactive state machines*, which users interact with by submitting transactions to a distributed ledger. A user submits a transaction, with the intention to issue some high-level command to the smart contract, e.g. to cast a vote, or withdraw funds. Once the transaction is confirmed by the distributed ledger, the user obtains information about the results of this high-level command: both whether it has been processed, and any information it may have computed using the contract's state.

As multiple users can interact with the same smart contract system concurrently, users cannot always predict the effect of their actions; a vote may end before a user's voting transaction is processed, for instance. As a result the user may not be able to predict the outcome of the command, or even if it can be carried out.

To capture privacy, the act of creating a transaction to post on the distributed ledger is the only point at which we permit privacy leakage. As a user may go offline at any point, any private information they reveal – a bid during an opening phase of an auction for instance – must be revealed in the on-chain transaction itself. Formally, we model this with a *leakage function* Λ, which describes what information is leaked if a user, seeing a specific contract state, issues a specific command. This function can also fix choices that an interaction may make – for instance if the command is "send a coin to Bob", it may decide *which* coin to send to Bob. To give users full control over their privacy, even when these decisions are complex or randomized, we ask them to sign off on a description of the leakage before the transaction is broadcast. The leakage in KACHINA captures information which a user purposely decides to reveal, as the functionality they gain by doing so is worth whatever damage they take to their private information. It is further worth noting that nothing prevents a malicious contract from finding clever ways to leak information without being observable. This highlights the importance of interacting only with trustworthy contracts, and the importance of the leakage descriptor being accurate.

Similarly to the leakage function, the semantics of the contract itself are largely dictated by a *transition function* Δ. It describes how the state of a smart contract evolves given a command and a few auxiliary inputs (such as the choice of coin alluded to above).

b) The core protocol idea: The KACHINA core protocol restricts itself to contracts which divide their state into a public state σ, and, for each party p, a private state ρ_p. These correspond to the shared ledger, and a party's local storage respectively. Transition functions are over pairs (σ, ρ_p) instead of over *all* private states – a party may only change their own private state. Honest users 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 [18, Appendix G], although it comes at the cost of increased public state sizes, and loss of anonymity).

A natural construction to achieve privacy in smart contracts utilizing zero-knowledge proof systems is apparent: On creating a transaction, a user p evaluates the transition function against the current contract state (σ, ρ_p), resulting in a state (σ', ρ'_p). He creates a zero-knowledge proof that $\sigma \mapsto \sigma'$ is a valid transition of public states (i.e. there exists a corresponding private state and input such that this transition takes place), and posts the proof and transition as a transaction. Locally, the user updates his private state to ρ'_p.

We can also clearly describe the leakage of this sketched protocol: The transition $\sigma \mapsto \sigma'$ is precisely the information which is revealed!

c) State oracles: The core protocol sketched above has two major problems:

1) Due to each transaction containing a proof of transition from one state to another, concurrent transactions will almost certainly fail once the state is changed.
2) The size of the statement being proved, and therefore the size of transactions, grows linearly with the overall size of the contract's state.

These drawbacks are especially notable in systems with many users and a high frequency of transactions: On Ethereum a transaction is almost certainly applied after many other transactions the author never knew about, nor should need to know about. The state the contract will be in once it executes a transaction, is something the transaction's author cannot predict accurately. In the naive system proofs only succeed in the state they were originally created for, as Figure 2 suggests. Instead of capturing a transition from $\sigma \mapsto \sigma'$, we would rather want to capture a (partial) function from states to successor states.

To solve these issues, we add a layer of indirection for accessing and updating contract states: Instead of the state being a direct input to the transition function, the contract has access to *oracles* operating on the public and private states. The contract makes queries to these oracles: functions which update the state, and return information about it. To prove

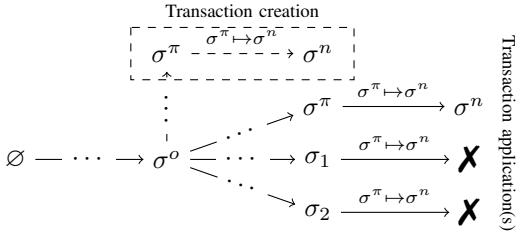

Fig. 2. Direct state-transition based transactions can be applied only in the state σ^π they were proven for.

the interaction with the public state correct, users capture the queries they made, and the responses they expect, in a sequence $((q_1, r_1), \ldots, (q_n, r_n))$: a *transcript* of oracle interactions. The user proves that, given the responses expected, they know an input which will make this series of queries.

Conversely, a user validating this transcript can verify this proof, and evaluate the queries in turn against the public state, ensuring the responses match. This defines a partial function over public states, which is defined wherever the responses recorded in the transcript match the results obtained by evaluating the queries on the current state.

Selecting what queries a contract makes provides a great deal of control over the domain of the function: a query which has an empty response will always succeed! In limiting queries to returning only essential information, many conflicts can be avoided. Transcripts can also be concise about what changes are made, assuming the queries are encoded in a sufficiently succinct language, such as most Turing-complete languages.

While not all conflicts are resolved through this as the responses may not match those expected, it allows the proof to focus on the *relevant* parts of the state, being compatible with more concurrent transactions, as pictured in Figure 3.

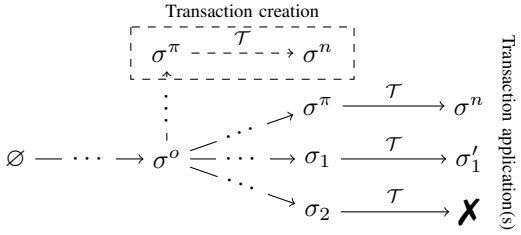

Fig. 3. Oracle-transcript based transactions can be applied in any compatible state. The transcript $\mathcal{T}$ defines a partial function $\{\sigma^\pi \mapsto \sigma^n, \sigma_1 \mapsto \sigma'_1, \ldots\}$.

In order to be able to model partial transaction success, which is crucial for modeling transaction fees, we allow for a special query to be made, COMMIT. COMMIT queries mark checkpoints in a transaction's execution, such that if an error occurs after it, the execution up to this point is still meaningful. This effectively partitions the transcript into atomic segments. We primarily use this to construct transaction fees within a smart contract itself, the details of which can be seen in [18, Appendix J.5].

d) High-level usage: Even when using state oracles, this protocol is limited to contracts which have their state fit neatly into accessing only shared public state, and local private state. The natural description of many contracts does not match this. For instance: a private currency contract is most directly described through a *shared private* state tracking the balances of all parties.

However, it *is* simple to express the Zerocash [1] protocol in terms of interactions with shared public, and local private states. This provides a practical means to *achieve* what we can *describe* using a shared private state. It is important to have both the most natural description of a contract, and the realization. The former provides a good understanding of the features and security properties of a contract, while the latter realizes it.

This idea is nothing but the notion of simulation-based security itself! We use multiple stages of UC-emulation: First moving from our objective contract (a private payments contract) to a contract within the KACHINA constraints on state (a Zerocash contract), and second moving on to the KACHINA core protocol. Due to the transitivity of UC emulation, we may therefore use this "KACHINA method" to construct the objective of private payments. This process is outlined in Figure 1.

Our model is designed to facilitate this usage. Specifically for modeling objective contracts the model allows the adversary to provide an additional adversarial input to each transaction. This input allows the simulator to control some parts of the ideal behaviour similar to the simulator's influence on an ideal functionality, for instance to ensure ideal world addresses match real-world public keys.

III. DEFINING SMART CONTRACTS

Smart contracts are typically implemented as replicated state machines. If a *replicated* state machine is the implementation, the natural model is that of the state machine itself. Inputs are drawn from a ledger of transactions, and passed to this state machine.

This definition is unsuitable for privacy-preserving smart contracts: If the state machine's behavior is known, and its inputs are on a ledger, there is no privacy. A simple tweak can solve this: Inputs are replaced with identifiers on the ledger, with the smart contract functionality tracking what their corresponding inputs are.

A. Interactive Automata Interpretation

Smart contracts are a form of *reactive computation*: Parties supply an input to the contract, the latter internally performs a stateful computation, and returns a result to the original caller. The result is returned asynchronously, and may depend on interactions with other users. This is quite close to the concept of a trusted third party, although real-world systems have caveats:

- They *leak* information about the computation performed.
- They allow some *adversarial influence*, partly due to relying on the transaction ordering of an underlying ledger.

- They may carry some impure *execution context*: A transaction may depend on what the state is at the time it is created, for instance.

Often when talking about smart contracts, only the "on-chain" component is considered. This is insufficient for privacy, as by its nature, everything on-chain is public. We therefore model the off-chain component of the interaction as well. This can be as simple as placing inputs directly on the ledger, but can involve more complex pre-computation. Even without the need for privacy, the need to model off-chain computation of smart contracts had been observed [9], and we believe a formal model should account for it.

To represent a contract, we use a transition function, operating over the contract's state. We denote the initial state as $\varnothing$. Transition functions are deterministic, although limited nondeterminism can be simulated by including randomness in the execution context. Notably, such randomness is fixed on transaction creation, allowing the creator to input (potentially biased) randomness, which is subsequently used in the (replicated) execution of the contract's state machine. Potential uses include the creation of randomized ciphertexts or commitments. The transition function will also output if a transaction should be considered "confirmed" or not, with the latter indicating failure or only partial success, which dependant transactions should not build on.

A contract *transition function* Δ is a pure, deterministic function with the format $(\phi', c, y) \leftarrow \Delta(\phi, p, w, z, a)$, with the following inputs and outputs:

- The current state ϕ
- The calling party p
- The input w
- The execution context z
- The adversarial input a
- The successor state ϕ'
- The output y
- The confirmation state c

In addition to the transition function, it is necessary to capture what leakage an interaction with the contract has. The two are separated due to the asynchronous nature of smart contracts – a transaction is made, and leaks information, before the corresponding transition function is run on-chain.

The leakage is captured by a *leakage function*, which receives the same input, and further receives the creating user p's "view" ω of the contract as an input. $\omega = (\ell, U_p, T, \phi)$ consists of four items: a) The length of p's view of the ledger ℓ. b) p's unconfirmed transactions U_p. c) A map T from $\tau \in U_p$ to (p, w, z, a, D). These are Δ's inputs, and the transaction's dependencies, which we will introduce shortly, D. d) The contract's state according to p's view of the ledger, ϕ. This "view" may be used to avoid attempting double-spends by selecting a coin to spend which no other unconfirmed transaction uses, for instance. For this purpose the leakage function can also abort by returning $\perp$, refusing to create a transaction. The function returns a leakage value lkg, which is passed to the adversary, a description of the leakage which occurred, desc, a list of transactions to depend on, D, and the context z. While lkg may be arbitrary, it is important that desc provides an accurate and readable description of this leakage. Its primary purpose is to allow parties to decide *not* to go ahead with a transaction if they notice the leakage is more than expected. With complex contracts, anticipating what will be leaked should not be relied upon. The usage of a descriptor highlights that Λ should not be maliciously supplied, and facilitates simulation, as shown in Section V.

It is worth emphasising that the leakage discussed in this paper is deliberate; this is not leakage observed over a network, which can be hard to identify, but is instead information which users accept to reveal. For instance, a leakage in Zerocash [1] is the length of the ledger at the time a transaction is created, with the security of the protocol guaranteeing that this – but nothing more – is revealed to an adversary.

The list of dependencies D is a list of transactions, which must occur in the same order before the newly created transaction can be applied. This can be used to enforce basic ordering constraints between transactions. Finally, the context z allows information about the state at the time of transaction creation to be passed to the transition function. This may include the current state, unconfirmed transactions, and a source of randomness. Its content is left arbitrary at this point.

A *leakage function* Λ is a pure, non-deterministic function with the format $(\mathsf{desc}, \mathsf{lkg}, D, z) \leftarrow \Lambda(\omega, p, w)$, with the following inputs and outputs:

- p's contract view ω
- The calling party p
- The input w
- The leaked data lkg
- The leakage descriptor desc
- The tx dependencies D
- The context z

We consider the pair (Δ, Λ) to define a smart contract. The ideal world interaction with a smart contract follows the below pattern:

1) A party submits a contract input w.
2) The corresponding context and leakage are computed.
3) The party agrees to the leakage description, or cancels (in the latter case, the transaction never takes place, and no information is revealed).
4) The adversary is given (lkg, D), and provides the adversarial input a.
5) The submitting party can retrieve the output of Δ (if any), while other parties can interact with the modified state.

The level of privacy guaranteed depends greatly on the leakage function Λ: A leakage function which returns its input directly as leakage provides no privacy, while one which returns no leakage at all provides almost total privacy (notably the fact some interaction was made is still leaked). By tuning this, the privacy of Ethereum, Zerocash, and everything in between can be captured.

Our model relies on users querying the result of transactions manually – they are not notified of the acceptance of a transaction, and can not modify it once made. If a transaction is not yet confirmed by the ledger, the user gets the result NOT-FOUND, if the transaction depends on failed transactions, $\perp$ is returned, and otherwise the result is provided by the contract itself (which may also inform of partial success).

B. UC Specification

The *ideal smart contract functionality* $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$ captures the notion of a contract as a leaky state machine whose inputs are drawn from a ledger. It is parameterized by the transition function Δ and the leakage function Λ, and it operates in a hybrid world with a global ledger functionality $\mathcal{G}_{\mathsf{ledger}}$. A candidate for such a ledger is $\mathcal{G}_{\mathsf{simpleLedger}}$, as provided in [18, Appendix B], although any compatible functionality is sufficient. Its privacy guarantees stem from only revealing explicitly leaked data, i.e. lkg, and only allowing the creator of a transaction to access the result.

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 $\phi \leftarrow \varnothing$, 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 $(\phi', c, y) \leftarrow \Delta(\phi, \ldots)$, $\phi' \neq \bot$. If both are satisfied, update ϕ to ϕ', and record the transaction as confirmed if c is $\top$. If an execution output is requested, return y, or $\bot$ if the execution failed. If, on the other hand, one of the preconditions is not satisfied, skip this transaction.

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

Retrieve p's current view ω of the contract. Feed this, together with the party identifier, 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 for p, send (SUBMIT, τ) to $\mathcal{G}_{\mathsf{ledger}}$, and finally return τ.

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

If τ is owned by p, and is in their current view of the ledger, compute and return the output by executing the ledger view up to τ. If τ is not in their ledger view, return NOT-FOUND.

IV. The Kachina Protocol

As mentioned in Section II, a naive construction divides a contract's state into a shared public state, and a local private states for each party. Specifically, the ideal state ϕ is defined as the tuple $(\sigma, \boldsymbol{\rho})$, where $\boldsymbol{\rho}$ consists of ρ_p for each party p. A user proves the validity of any public state transition – that there exists a private state and input, such that this transition takes place. This clearly does not scale well, as it assumes that the ledger state does not change between the submission and processing of a transaction, and requires zero-knowledge proofs about potentially large states – hundreds of Gigabytes in systems like Ethereum [13]!

In reality, a user's query may not be evaluated immediately, and the ledger may change drastically in the meantime. Simply proving a direct state transition would lead to a high proportion of queries being rejected. To solve both problems, we require contracts to access their state through a layer of abstraction which both tolerates reordering interactions, and allows for more efficient proofs. We further allow for partial transaction success, by introducing *transaction checkpoints*. Our primary purpose for this notion is to be able to capture the payment of transaction fees, such as gas. We detail our approach to do this in [18, Appendix J.5].

A. State Oracles and Transcripts

We introduce *state oracles* and *state oracle transcripts* to abstract interaction with a contract's state. We choose this abstraction primarily for its flexibility, and many other approaches are possible, such as byte-level memory accesses, or specific data structures such as set of unspent transactions. These can be seen as instances of state oracles. We make use of the notation $[a, b, c]$ to denote a list of a, b, and c, with the concatenation operator $\|$, and the empty list ϵ. We use the function last to retrieve the last element of a list, and $L[i]$ to denote the ith element of the list L.

a) An example: To better motivate the need to abstract interactions with a contract's state, we will use a representative example smart contract, and discuss how different abstractions of its state will affect it.

Our example is a *sealed bid auction* contract[1], which we assume has some means of interacting with two on-chain assets, one public and one private. These may be constructed similarly as in Section V, 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 [4]. The auction is opened by the *seller* party, and multiple *buyer* parties may bid on it. The auction has three stages: Bidding, opening, and withdrawing. The auction contract allows for the following interactions:

- At initialization, the seller transfers ownership of the public asset A to the auction contract.
- In Stage 1, buyers submit their bids, transferring some amount of the private asset B to the auction contract, which remains anonymous.
- In Stage 2, buyers *reveal* their bid. If the buyer's bid exceeds the currently maximum revealed bid, they reveal their committed asset, increase the maximum bid, and they record themselves as the winning bidder. Otherwise, they withdraw their bid from the contract without revealing its value.
- In Stage 3, buyers withdraw any assets they own after the auction – either their (losing) bids, or the sold asset (for

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

the highest bidder). The seller withdraws the highest bid, or the original asset if no bids were made.
- In Stage 1 and 2, the seller may advance the stage.

This contract needs to maintain in its state:

- The current stage the auction is in.
- A reference to the asset being sold.
- A set of bids made.
- The winning bid, its value, and who made it, during the reveal phase.
- A set of losing bids, which have not yet been withdrawn, during the reveal phase.
- Privately, a user remembers which bids are theirs, and how to reveal them.

Suppose we adopted a naive 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 succeed – as soon as this transaction changes the state by adding its own bid, the proof of any other simultaneous transaction becomes invalid.

The simple abstraction of byte-level access would allow a buyer and a seller to withdraw concurrently, as their withdrawals affect different parts of the state. It does not do so well in allowing concurrent bids to be made, however. If the set is implemented with a linked list, for instance, two users attempting to add their own bid simultaneously will change the same part of the state: the pointer to the next element.

A smart abstraction should realize that whichever user bids first, the resulting set of bids is the same, even if its binary representation may not be. Even if the order of the interactions matters, a smart abstraction may allow concurrent interactions. When claiming the maximum bid in the auction, Alice may increase it to 5, while Bob may increase it to 7 concurrently. It should not matter to Bob's transaction if the maximum bid is currently 3, or 5 – although Alice's must be rejected if the bid is increased to 7 first.

b) General-purpose state oracles: The abstraction we propose is that of *programs*. Appending a value to an linked list can be encoded as a program which a) traverses to the end of the current list, b) creates a new cell with the input value, and c) links this from the end of the list. Formally, these programs are executed by a universal machine called a *state oracle* with access to the current (public or private) state α, and potentially an additional *context* z.

Definition 1. A state oracle $\mathcal{O} = \mathcal{U}(\alpha_0, z)$, given an initial state α_0, and context z, is an interactive machine internally maintaining a state α, a transcript $\mathcal{T}$, and a vector of fallback states $\vec{\boldsymbol{\alpha}}$ (initially set to the input α_0, ϵ, and $[\alpha_0]$ respectively), which permits the following interactions:

- Given a COMMIT query, set $\vec{\boldsymbol{\alpha}} \leftarrow \vec{\boldsymbol{\alpha}} \parallel [\alpha]$, and append COMMIT to $\mathcal{T}$.
- Given a query q while α is $\perp$, return $\perp$.
- Otherwise, given a query q, compute $(\alpha', r) \leftarrow q(\alpha, z)$. Update α to α', append (q, r) to $\mathcal{T}$, and return r.
- $\texttt{state}(\mathcal{O})$ returns $(\vec{\boldsymbol{\alpha}} \parallel [\alpha], \mathcal{T})$.

The context z is empty ($\varnothing$) for state oracles operating on the public state, and is used in state oracles operating on the private state for fine-grained read-only access to the state during transaction creation, e.g. to allow private state oracles to read the public state. Specifically, the oracle operating on the private state can read both the public and private states for: a) the confirmed state at the time the transaction was created (σ^o and ρ^o), and b) the *projected* state, an optimistic state in which all of the user's unconfirmed transactions are executed, at the time the transaction was created (σ^π and ρ^π). This can be used to make sure new transactions do not conflict with pending ones: Selecting which coin to spend uses the confirmed state to ensure the coin *can be spent*, and the projected state to ensure a coin is not *double spent*. The context is also used to provide a source of randomness η to the private state oracle. In total, the context of the private state oracle is $(\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta)$. The context to the public state oracle is empty ($\varnothing$), and we will sometimes omit it.

We say that the oracle *aborts* if it sets its state to $\perp$. The state will then be rolled back to a safe point, specifically the last COMMIT where the state was non-$\perp$. Looking forward, we will decompose the transition function Δ into three components: An oracle operating on the public state σ, an oracle operating on p's private state ρ_p, and a "core" transition function Γ. This process is described in detail in Subsection IV-D, with an overview of the interactions of Γ with public and private state oracles given in Figure 4.

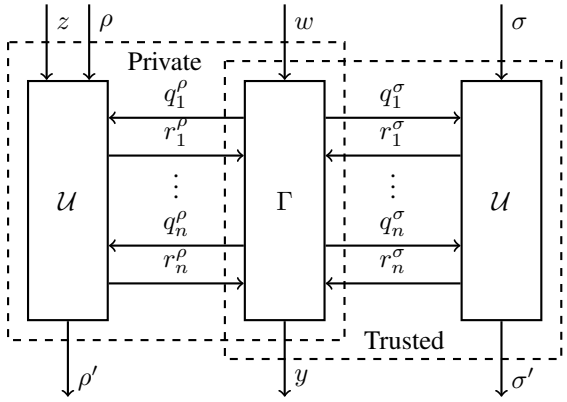

Fig. 4. The interaction of the core contract Γ, with two universal machines $\mathcal{U}$, acting as state oracles over the public state σ, and the private state ρ, together with the context z.

The notion of *oracle transcripts* 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: We only prove that given some input, and a sequence of responses recorded in the public state transcript, the smart contract must have made the recorded queries.

c) Revisiting our example: As an illustration, we show how our auction example interacts with state oracles. We define the auction's states more precisely first, where users are identified by public keys, denoted with pk:

- The current stage, $\mathsf{stage} \in \{1, 2, 3\}$.
- A reference to the asset being sold and who is selling it: a, pk_s.
- A set of bids made S.
- The winning bid, its value, and who made it: b, v, pk_b.
- A set of not yet withdrawn losing bids: R.
- Privately, a user remembers openings to their bids, the committed bid itself, and its value: $\mathsf{bidOpen}, \mathsf{bidComm}, v$.

Overall, the public state is defined as $\sigma := (\mathsf{stage}, \mathsf{pk}_s, a, b, v, \mathsf{pk}_b, S, R)$, and the private state is defined as $\rho := (\mathsf{bidOpen}, \mathsf{bidComm}, v)$. The public state is initialized by the seller to $(1, \mathsf{pk}_s, a, \varnothing, 0, \varnothing, \varnothing, \varnothing)$.

The oracle queries corresponding to each interaction with the contract are given as closures, i.e. sub-functions which make use of some of their parents local variables. To clarify where this is the case, we place such variables in the subscript of the function name. These functions are passed to either the public or private state oracle as the input q, as specified in Definition 1.

- Bidding: Given a asset opening $\mathsf{bidOpen}$, with value v, corresponding to an asset commitment $\mathsf{bidComm}$, which has been bound to the auction contract, Γ first makes the following public oracle query:

  function $\mathsf{makeBid}_{\mathsf{bidComm}}((\mathsf{stage}, \mathsf{pk}_s, a, b, v, \mathsf{pk}_b, S, R))$
  assert $\mathsf{stage} = 1$
  return $((\mathsf{stage}, \mathsf{pk}_s, a, b, v, \mathsf{pk}_b, S \cup \{\mathsf{bidComm}\}, R), \top)$

  Further, it makes the following private oracle query:

  function $\mathsf{recordBid}_{\mathsf{bidOpen},\mathsf{bidComm},v}(\cdot, \cdot)$
  return $((\mathsf{bidOpen}, \mathsf{bidComm}, v), \top)$

- 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 $\mathsf{retrieveBid}((\mathsf{bidOpen}, \mathsf{bidComm}, v), \cdot)$
  return $((\mathsf{bidOpen}, \mathsf{bidComm}, v), (\mathsf{bidOpen}, \mathsf{bidComm}, v))$

  Next, the contract makes a further private oracle query for the expected maximum bid, to determine if the buyer's bid is higher:

  function $\mathsf{projMax}(\rho, z = (\cdot, \cdot, \sigma^\pi = (\ldots, v', \ldots), \cdot, \cdot))$
  return (ρ, v')

  If this query returns $v' < v$, the contract attempts to claim the maximum bid with the public oracle query[2]:

  function $\mathsf{claimMax}_{\mathsf{bidOpen},\mathsf{bidComm},v,\mathsf{pk}}(\sigma)$
  let $(\mathsf{stage}, \mathsf{pk}_s, a, b_o, v_o, \mathsf{pk}_o, S, R) \leftarrow \sigma$
  assert $\mathsf{bidComm} \in S \wedge v > v_o \wedge \mathsf{stage} = 2$
  return $((\mathsf{stage}, \mathsf{pk}_s, a, \mathsf{bidOpen}, v, \mathsf{pk}, S \setminus \{\mathsf{bidComm}\}, R \cup \{(b_o, \mathsf{pk}_o)\}), \top)$

  If the original value test fails, on the other hand, instead the contract transfers the ownership of $\mathsf{bidComm}$ via the underlying asset system to pk, and runs the public oracle query:

  function $\mathsf{claimLoss}_{\mathsf{bidComm}}((\mathsf{stage}, \mathsf{pk}_s, a, b, v_o, \mathsf{pk}_o, S, R))$
  assert $\mathsf{bidComm} \in S \wedge \mathsf{stage} = 2$
  return $(\top, (\mathsf{stage}, \mathsf{pk}_s, a, b, v_o, \mathsf{pk}_o, S \setminus \{\mathsf{bidComm}\}, R))$

- 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 in a public oracle query:

  function $\mathsf{withdraw}_{\mathsf{pk}}((\mathsf{stage}, \mathsf{pk}_s, a, b, v, \mathsf{pk}_b, S, R))$
  assert $\mathsf{stage} = 3$
  if $\mathsf{pk} = \mathsf{pk}_s \wedge b \neq \varnothing$ **then**
  return $((\mathsf{stage}, \varnothing, a, \varnothing, \varnothing, \mathsf{pk}_b, S, R), (B, b))$
  else if $\mathsf{pk} = \mathsf{pk}_b \wedge a \neq \varnothing$ **then**
  return $((\mathsf{stage}, \mathsf{pk}_s, \varnothing, b, v, \varnothing, S, R), (A, a))$
  else if $\exists c : (c, \mathsf{pk}) \in R$ **then**
  return $((\mathsf{stage}, \mathsf{pk}_s, a, b, v, \mathsf{pk}_b, S, R \setminus \{(c, \mathsf{pk})\}), (B, c))$

- Advancing the stage: The seller (given their public key pk) may advance the contracts stage from 1 or 2 to 2 or 3 respectively with a public oracle query:

  function $\mathsf{advanceStage}_{\mathsf{pk}}((\mathsf{stage}, \mathsf{pk}_s, a, b, v, \mathsf{pk}_b, S, R))$
  assert $\mathsf{pk} = \mathsf{pk}_s \wedge \mathsf{stage} \in \{1, 2\}$
  return $((\mathsf{stage} + 1, \mathsf{pk}_s, a, b, v, \mathsf{pk}_b, S, R), \top)$

This example does not handle corner cases (such as buyers bidding multiple times), and is not intended for practical use. Instead, its purpose is to illustrate the advantages state oracles provide: The query an interaction will make, and the response it will receive, are often not affected by other interactions. Concurrent bids do not conflict, for instance. The representation of data is also not crucial, as the state oracles may themselves interact with abstract data types.

We complete our example by specifying the core transition function Γ, under the assumptions that a means to call into a separate asset management system (a contract that permits transferring ownership of assets between public keys), such as presented in [18, Appendix J.4], exists. We also assume that a user's public key can be retrieved with a shared "identity" contract.

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

A simple private auction contract.

When receiving an input (BID, v)*:*
send $(\text{BIND}, v, \Gamma_{\mathsf{auction}})$ **to** Γ_B and
receive the reply $(\mathsf{bidOpen}, \mathsf{bidComm}, v)$
send $\mathsf{makeBid}_{\mathsf{bidComm}}$ **to** $\mathcal{O}_\sigma$ and **receive the reply** $\top$
send $\mathsf{recordBid}_{\mathsf{bidOpen},\mathsf{bidComm},v}$ **to** $\mathcal{O}_\rho$ and
receive the reply $\top$
When receiving an input REVEAL*:*
send $\mathsf{retrieveBid}$ **to** $\mathcal{O}_\rho$ and
receive the reply $(\mathsf{bidOpen}, \mathsf{bidComm}, v)$
send IDENTITY **to** Γ_{id} and **receive the reply** pk
send $\mathsf{projMax}$ **to** $\mathcal{O}_\rho$ and **receive the reply** v'
if $v' < v$ **then**

[2]Note that the claim may fail if the maximum bid increased from the one projected at the time of transaction creation.

send (ASSERTVALIDFOR, bidOpen, bidComm, v, pk, Γ_{auction}) **to** Γ_B
send $\text{claimMax}_{\text{bidOpen,bidComm},v,\text{pk}}$ **to** $\mathcal{O}_\sigma$ and **receive the reply** $\top$
else
send (UNBIND, bidOpen, pk) **to** Γ_B
send $\text{claimLoss}_{\text{bidComm}}$ **to** $\mathcal{O}_\sigma$ and **receive the reply** $\top$

When receiving an input WITHDRAW*:*

send IDENTITY **to** Γ_{id} and **receive the reply** pk
send $\text{withdraw}_{\text{pk}}$ **to** $\mathcal{O}_\sigma$ and **receive the reply** (X, x)
if $X = A$ **then**
send (TRANSFER, x, pk) **to** Γ_A
else
send (UNBIND, x, pk) **to** Γ_B

When receiving an input ADVANCE-STAGE*:*

send IDENTITY **to** Γ_{id} and **receive the reply** pk
send $\text{advanceStage}_{\text{pk}}$ **to** $\mathcal{O}_\sigma$ and **receive the reply** $\top$

d) Using transcripts: KACHINA relies on a few key observations on how transcripts relate to the original state oracle execution. To begin with, we define a few ways in which transcripts may be used.

Definition 2. A state oracle transcript $\mathcal{T}$ may be applied to a state α in a context z. We write $\vec{\boldsymbol{\alpha}} \leftarrow \mathcal{T}(\alpha, z)$, or if $z = \varnothing$, $\vec{\boldsymbol{\alpha}} \leftarrow \mathcal{T}(\alpha)$, the operation of which is defined through the following loop:

function $\mathcal{T}(\alpha, z)$
let $\mathcal{O} \leftarrow \mathcal{U}(\alpha, z)$
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 $(\vec{\boldsymbol{\alpha}}, \cdot) \leftarrow \mathsf{state}(\mathcal{O})$
return $\vec{\boldsymbol{\alpha}}$

If a transcript is malformed, applying it will result in $[\alpha, \bot]$.

The returned $\vec{\boldsymbol{\alpha}}'$ is indistinguishable from the internal state $\vec{\boldsymbol{\alpha}} \parallel [\alpha]$ of the state oracle $\mathcal{U}(\alpha_0, z)$, given the same sequence of queries. This allows users to replicate the effect other users' queries have on the public state, without knowing *why* these queries were made.

Definition 3. A sequence of transcripts and contexts $X = ((\mathcal{T}_1, z_1), \ldots, (\mathcal{T}_n, z_n))$ is applied by applying each transcript in order. We write $\mathcal{T}^*_X(\alpha)$, which has the recursive definition:

- $\mathcal{T}^*_\epsilon(\alpha) := \alpha$
- $\mathcal{T}^*_{X \parallel [(\mathcal{T},z)]}(\alpha) := \mathsf{last}(\mathcal{T}(\mathcal{T}^*_X(\alpha), z))$

Definition 4. A transcript $\mathcal{T} = ((q_1, r_1), \ldots, (q_n, r_n))$ (potentially including COMMIT messages) induces a transcript oracle $\mathcal{O}(\mathcal{T})$, which behaves as follows:

- Recorded COMMIT messages are ignored.
- For the ith query q'_i, return r_i if $q'_i = q_i$, otherwise abort by returning $\bot$ for this, and all subsequent queries.
- When $\mathsf{consumed}(\mathcal{O})$ is queried, return $\top$ if exactly n queries were made, otherwise return $\bot$.

If in an interaction with the oracle, consumed holds, the transcript was *minimal* for this interaction.

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 behavior is correct, *given the responses the transcript claims*. We also prove that $\mathsf{consumed}(\mathcal{O})$ holds, ensuring the transcript doesn't just start with the queries an honest execution would make, but that it matches them exactly.

These are used together to define how a transaction is made, and how it is applied: Alice generates a transcript for the oracle accesses her transaction will perform, and proves this transcript both correct and minimal. She sends the transcript and proof to Bob, who is convinced by the proof of correctness and minimality, and can therefore reproduce the effect of the transaction by applying the transcript to the state directly.

e) Inherent conflicts: Abstracting the interaction with the state has many benefits, but it is not a panacea. Some conflicts are inherent, and unavoidable – a contract may operate on a first-come first-serve basis, and no trick will ease the pain of coming second. A contract may also simply be badly designed, not making good use of the abstractions provided – at the most extreme, it can make only queries retrieving or setting the entire state, negating all benefit of using oracles.

B. Interaction Between Smart Contracts

The example in Subsection IV-A, makes the natural assumption (in the setting of smart contracts), of being able to interact with other components – in this case with an asset system. Most interesting applications of smart contracts seem premised on such interactions. We consider how multiple contracts may interact in [18, Appendix J.3], however we stress that a full treatment is left as future work.

In particular, how various contracts can be independently proven secure and composed in a general system alongside other, potentially malicious contracts, is not handled in this paper. Instead, where we assume interaction, we limit ourselves to a closed smart contract system with a small set of non-malicious contracts, such as the auction contract and the asset system in Subsection IV-A.

While it is tempting to delegate such interactions to the native compositionality and interactiveness of UC, this does not reflect the reality of smart contract interactions, where the executions of multiple contracts are atomically intertwined. While related issues of interaction with the environment have been considered in the literature, for instance in [6, 5], they do not fully address our scenario, in which multiple branches can be executed in projection. We therefore believe that studying the interaction and composition of smart contract transition and leakage functions requires further work, with this work providing a foundation.

C. The Challenge of Dependencies

If a transaction τ_1 moves funds from Alice to Bob, and τ_2 moves funds from Bob to Charlie, the order $\tau_2 \ldots \tau_1$ may not be valid, if τ_2 relies on the funds Bob received from Alice. When a dependency like this is violated in interacting with

the public state, attempting to apply the dependent transaction first will fail, and the transaction is rejected.

How such interactions affect a user's private state is more tricky to handle. While two different parties cannot conflict with each other on private state changes due to domain separation, parties may encounter *internal* dependencies.

A party starting with the private state ρ_1, makes a transaction τ_1 which advances their private state to ρ_2. Afterwards, they make the transaction τ_2, their private state ending up as ρ_3. If these transactions are made shortly after each other, τ_2 may be placed before τ_1 on the ledger. It is possible that τ_2 uses information from τ_1, such as a secret key, and that it makes no sense without it.

Should a user ignore the reordering, and stick with the state ρ_3? This can introduce inconsistencies between the public state and private state. Should the user apply the private state transcript of τ_2 and hope for the best – but risk a catastrophic failure if it cannot be applied? Neither are ideal. Instead, we propose that τ_2 should publicly declare that it depends on τ_1, and rely on on-chain validation to ensure they are applied in the correct order.

If a user has a set of unconfirmed transactions U, and is adding the new transaction τ in the ledger state, dependencies should ensure that any permutation of $U \cup \{\tau\}$ results in a consistent interaction with the user's private state – i.e. result in a non-$\bot$ private state. Further, this should even be the case if these transactions are only partially successful – regardless as to which COMMIT point was reached.

An overeager approach would be to ensure all unconfirmed transactions are dependencies, and in the order that they were made. With domain separation and sufficiently abstract interactions it is likely that only few transactions actually depend on each other. This can be application specific, and to account for this we allow for contracts to specify a function dep to declare dependencies. We constrain how this function may behave, and provide the all-purpose fallback of all unconfirmed transactions.

For most practical cases that we have observed, private state oracles do not conflict or enter into complex dependencies with each other. Most often, their state management is simple: sampling and storing secrets. The formal machinery presented in this section is to allow this intuition that the transactions do not depend on each other to be justified in many cases.

a) Formal definition: The formal definition of dependency functions is complex; we begin by introducing some mathematical notations. In addition to this notation, we make use of the following functions: a) the higher-order function map. b) an index function, which returns the index of an element in a list, idx. c) the tuple projection functions proj_i, which return the ith element of a tuple. d) the list flattening function $\mathsf{flatten}$, which, given a list of lists, returns a list of the inner lists concatenated. e) the function take, which returns the prefix of a list containing a specified number of items. f) the function zip, which combines n lists into a list of n-tuples.

Definition 5. For any finite set X, S_X is the set of all permutations of X, where each permutation is a list.

Definition 6. The *subsequence relation* $X \sqsubseteq Y$ indicates that each element of the list X is present in Y, in the same order:

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

We define an expansion of transactions into useful components: As a transaction has no private data within it, we use this to refer to this data.

Definition 7. A transcript $\mathcal{T}$'s corresponding *commit-separated* transcript $\vec{\mathcal{T}}$ is a list of lists of query/response pairs, corresponding to splitting $\mathcal{T}$ at each COMMIT. We write $\vec{\mathcal{T}} = \mathsf{split}(\mathcal{T}, \text{COMMIT})$.

Definition 8. A secret-expanded transaction is a tuple $(\tau, \vec{\mathcal{T}}, z, D)$, consisting of the transaction object τ, the commit-separated private state transcript $\vec{\mathcal{T}}$, the context z, and the dependencies D.

We define the format of transactions handled by the dependency function. We make use of "confirmation depth", the vector of which is denoted $\vec{c}$. This is a vector of natural numbers, representing how many parts of the corresponding commit-separated transcript executed successfully.

Definition 9. A list X of secret-expanded transactions' dependencies may be *satisfied* given a set of still unconfirmed transaction U and a list of confirmation depths $\vec{c}$, denoted by $\mathsf{sat}(X, \vec{c}, U)$, which is defined formally below. Informally, it states that each transaction in X must be preceeded by its dependencies, in order, and that each of these dependencies should have executed fully, rather than partially.

- $\mathsf{sat}(\epsilon, \vec{c}, U) := \top$
- $\mathsf{sat}(X \parallel (\cdot, \cdot, \cdot, D), \vec{c} \parallel \cdot, U) := \mathsf{sat}(X, \vec{c}, U) \wedge (D \cap U) \sqsubseteq \mathsf{map}(\mathsf{proj}_1, X) \wedge \forall d \in D, \vec{\mathcal{T}}, z, D', i : (d, \vec{\mathcal{T}}, z, D') = X[i] \implies |\vec{\mathcal{T}}| = \vec{c}[i]$

We write $\mathsf{sat}^*(X, U)$ as a shorthand for the case where $\vec{c}$ are maximal: i.e. $\vec{c}[i] = |\mathsf{proj}_2(X[i])|$.

We define what transcripts will actually be executed for a given sequence of confirmation levels.

Definition 10. The *effective sequence of transcripts* (denoted $ES(X, \vec{c})$), given a list of secret-expanded transactions and a list of confirmation depths of equal length, is the sequence of confirmed transcript parts, along with their contexts, defined as:

$ES(X, \vec{c}) := \mathsf{flatten}(\mathsf{map}(\lambda((\cdot, \vec{\mathcal{T}}, z, \cdot), c) : \mathsf{map}(\lambda\mathcal{T} : (\mathcal{T}, z), \mathsf{take}(\vec{\mathcal{T}}, c)), \mathsf{zip}(X, \vec{c})))$

We write $ES^*(X)$ as a shorthand for the case where $\vec{c}$ are maximal: i.e. $\mathsf{proj}_i(\vec{c}) = |\mathsf{proj}_2(\mathsf{proj}_i(X))|$.

We define the central invariant the dependencies must preserve: That the private state can always be advanced.

Definition 11. The dependency invariant $J(X, \rho)$, given a set X of secret-expanded transactions, states that any permutation of a subset of X's private state transcripts which

have their dependencies satisfied can be successfully applied to ρ. $J(X, \rho) := \forall Y \subseteq X, Z \in S_Y, \vec{\boldsymbol{c}} : \mathsf{sat}(Z, \vec{\boldsymbol{c}}, \mathsf{map}(\mathsf{proj}_1, X)) \implies \mathcal{T}^*_{ES(Z,\vec{\boldsymbol{c}})}(\rho) \neq \bot$

Finally, we define the constraints on the dependency function.

Definition 12. A dependency function $\mathsf{dep}(X, \mathcal{T}, z)$ is a pure function taking as inputs a set of secret-expanded unconfirmed transactions X, a new private state transcript $\mathcal{T}$, and a new context z, returning a list of transaction objects. It must satisfy the following conditions:

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 private state transcript $\mathcal{T}$ (where its commit-separated form is $\vec{\boldsymbol{\mathcal{T}}}$) and context z, the dependency invariant J is preserved: **let** $Y = X \parallel (\tau, \vec{\boldsymbol{\mathcal{T}}}, z = (\cdot, \rho^o, \cdot, \cdot, \cdot), \mathsf{dep}(X, \mathcal{T}, z))$ **in** $\mathcal{T}^*_{ES^*(Y)}(\rho^o) \neq \bot \wedge J(X, \rho^o) \implies J(Y, \rho^o)$

The dependency function $\mathsf{dep}(X, \mathcal{T}, z) = \mathsf{map}(\mathsf{proj}_1, X)$ can always be used, as it maximally constraints the possible permutations which satisfy dependencies.

D. The Contract Class

The core KACHINA protocol can realize a class of smart contracts, with each contract being primarily defined by a restricted transition function Γ. This transition function is given oracle access to the calling user's private state ρ_p and the shared public state σ, as described in Definition 1. In addition to these oracle accesses, Γ can make (COMMIT, y) queries, which a) send COMMIT to both oracles, and b) record the value y in a vector of partial results $\vec{\boldsymbol{y}}$. We write $\vec{\boldsymbol{y}} \leftarrow \Gamma_{\mathcal{O}_\sigma, \mathcal{O}_\rho}(w)$ as running the transition function against input w, with oracles $\mathcal{O}_\sigma$ and $\mathcal{O}_\rho$, returning the vector of partial results $\vec{\boldsymbol{y}}$. The final output of Γ is appended to $\vec{\boldsymbol{y}}$ when it returns. The adversary can program its own private state oracle – it corresponds to local computation, after all! Two minor functions are also used to define the corresponding ideal contract:

- The leakage descriptor desc, which receives the time t, the sequence of secret-expanded unconfirmed transactions X, transcripts $\mathcal{T}_\sigma, \mathcal{T}_\rho$, original input w, and context z of new transactions as inputs, and returns a description of what leakage this interaction will incur.
- A dependency function dep satisfying Definition 12.

Definition 13. $\mathbb{C}_{\text{KACHINA}}$ is the set of all pairs $(\Delta_{\text{KACHINA}}(\Gamma), \Lambda_{\text{KACHINA}}(\Gamma, \mathsf{desc}, \mathsf{dep}))$, for any parameters Γ, desc and dep, satisfying Definition 12.

Δ_{KACHINA} and Λ_{KACHINA} operate as follows, with a full description in [18, Appendix C]. We assume the set of honest parties $\mathcal{H}$ – in the ideal world, this is known by the functionality, while in the real world we assume each party p will use $\mathcal{H} = \{p\}$.

Transition Function $\Delta_{\text{KACHINA}}(\Gamma)$ (sketch)

When receiving an input $((\sigma, \boldsymbol{\rho}), p, w, (\mathcal{T}_\sigma, z), \cdot)$*:*
let $(\vec{\boldsymbol{\sigma}}, \mathcal{T}'_\sigma, \vec{\boldsymbol{\rho}}, \cdot, \vec{\boldsymbol{y}}) \leftarrow \mathsf{run}\text{-}\Gamma(\sigma, \boldsymbol{\rho}[p], w, z, p \in \mathcal{H})$
let $\sigma' \leftarrow \sigma; y \leftarrow \bot; C \leftarrow \top$
let $\vec{\boldsymbol{\mathcal{T}}} \leftarrow \mathsf{split}(\mathcal{T}_\sigma, \text{COMMIT}); \vec{\boldsymbol{\mathcal{T}}}' \leftarrow \mathsf{split}(\mathcal{T}_\sigma, \text{COMMIT})$
for $(\mathcal{T}_i, \mathcal{T}_c, \sigma', \rho', y')$ **in** $\mathsf{zip}(\vec{\boldsymbol{\mathcal{T}}}, \vec{\boldsymbol{\mathcal{T}}}', \vec{\boldsymbol{\sigma}}, \vec{\boldsymbol{\rho}}, \vec{\boldsymbol{y}})$ **do**
  if $\sigma' = \bot \vee \rho' = \bot \vee \mathcal{T}_r \neq \mathcal{T}_c$ **then**
    let $C \leftarrow \bot$
    break
  let $\sigma \leftarrow \sigma'; \boldsymbol{\rho}[p] \leftarrow \rho'; y \leftarrow y'$
return $((\sigma, \boldsymbol{\rho}), C, y)$
Where $\mathsf{run}\text{-}\Gamma(\sigma, \rho, w, z, \cdot)$ runs $\Gamma_{\mathcal{O}_\sigma, \mathcal{O}_\rho}(w)$, and returns $(\vec{\boldsymbol{\sigma}}, \mathcal{T}_\sigma, \vec{\boldsymbol{\rho}}, \mathcal{T}_\rho, \vec{\boldsymbol{y}})$ (see [18, Appendix C] for a full specification).

Leakage Function $\Lambda_{\text{KACHINA}}(\Gamma, \mathsf{desc}, \mathsf{dep})$ (sketch)

When receiving an input $(\omega = (\ell, U, T, \phi = (\sigma^o, \boldsymbol{\rho}^o)), p, w)$*:*
Simulate applying all unconfirmed transactions in order, for a new *projected state* $(\sigma^\pi, \boldsymbol{\rho}^\pi)$. Select a randomness stream η, and set the context z to the old state $(\sigma^o, \boldsymbol{\rho}^o[p])$, the projected state $(\sigma^\pi, \boldsymbol{\rho}^\pi[p])$, 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 $(\mathsf{description}, \mathcal{T}_\sigma, D, (\mathcal{T}_\sigma, z))$.

E. The Core KACHINA Protocol

The construction of the core protocol itself is now fairly straightforward. We use non-interactive zero-knowledge to prove statements about transition functions interacting with an oracle. 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 [18, Appendix C].

Formally, the language $\mathcal{L}$ of the NIZK used is defined as follows, for any given transition function Γ: $((\mathcal{T}_\sigma, \cdot), (w, \mathcal{T}_\rho)) \in \mathcal{L}$ if and only if, where $\mathcal{O}_\sigma \leftarrow \mathcal{O}(\mathcal{T}_\sigma)$, and $\mathcal{O}_\rho \leftarrow \mathcal{O}(\mathcal{T}_\rho)$, $\mathsf{last}(\Gamma_{\mathcal{O}_\sigma, \mathcal{O}_\rho}(w)) \neq \bot$, and after it is run, $\mathsf{consumed}(\mathcal{O}_\sigma) \wedge \mathsf{consumed}(\mathcal{O}_\rho)$ holds. This is efficiently provable provided that $\mathcal{T}_\sigma$, w, and $\mathcal{T}_\rho$ are short, and Γ itself is efficiently expressible in the underlying zero-knowledge system.

Protocol KACHINA (sketch)

The KACHINA protocol realizes the ideal smart contract functionality when parameterized by a transition function Γ, a leakage descriptor desc, and a dependency function dep, satisfying Definition 12. It operates in the $(\mathcal{F}^{\mathcal{L}}_{\mathsf{nizk}}, \mathcal{G}_{\mathsf{simpleLedger}})$-hybrid model.

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 proofs. If they are satisfied, apply $\mathcal{T}_\sigma$ in commit-separated parts, up to (not including) the first $\bot$

result, if any. If available, execute $\mathcal{T}_\rho$ to the same depth, and if this depth is the full depth of the transcript, mark the transaction as confirmed. If an output is requested, and the transaction's output vector $\vec{\boldsymbol{y}}$ is available, return the output indexed with the confirmation depth. 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 vector $\vec{\boldsymbol{y}}$. Compute the dependencies D and leakage description description.

Ask p if 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 vector $\vec{\boldsymbol{y}}$, and publish $\tau = (\mathcal{T}_\sigma, D, \pi)$ on $\mathcal{G}_{\mathsf{ledger}}$. Record τ as unconfirmed, and return it.

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 to retrieve the output associated with τ, if any.

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 [18, Appendix D]. 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 J of Definition 11. 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 preserved, 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 straightforward; 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 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.

V. A Case Study: Private Payments

To demonstrate the versatility of KACHINA, we take a closer look at the (private) token contract, which is prone to the scalability issues KACHINA addresses. Public token contracts are well understood, and standardized [27], with the typical implementation being to maintain a mapping of "addresses" (hashes of public keys) to balances in the contract's public state. We write the first *provably private* token contract to demonstrate the expressive power of KACHINA.

A private token contract also implies that currency is not a primitive – it can be built as a contract, a key factor in simplifying our model, as it does not need to encode currency as a special case. It provides an asset to build contracts around in the first place, as well as a means of denial-of-service mitigation, through transaction fees. Bad fee models have resulted in devastating DoS attacks [29], highlighting their necessity.

We detail how to construct a fee model in [18, Appendix J.5]. The fundamental idea of this construction is to embed the transition function Γ in a wrapper which performs the following steps:

1) In the private state oracle, estimate the cost of transaction fees.
2) Given an input gas price, and this estimate, pay these fees using a designated currency contract.
3) Commit this as a partial execution success.
4) Execute Γ with a modified $\mathcal{O}_\sigma$, which deducts from available gas for each operation and aborts if this runs out.
5) Transfer any remaining gas back to the transaction author.

A. Indirect Construction

Following the design of Zerocash [1], we write a contract that maintains the necessary Zerocash secrets: coin randomnesses, commitment openings, and secret keys. The private state oracle computes 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 is 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.

This design is not self-evidently correct, and is not the objective itself. Specifying what goal it achieves, in terms of an ideal leakage and transition function, allows us to build a clean ideal world, with a clear private token contract. This ideal world is constructed in two steps: First showing that the

Zerocash contract UC-emulates it, and second showing that the Zerocash contract is in turn UC-emulated by KACHINA.

B. Ideal Private Payments

To simplify the external interface, we only use single denomination coins. The same approach can be applied to the full Zerocash protocol, with some caveats on coin selection and leakage.

We formally specify the private token contract through its transition and leakage functions, Δ_{pp} and Λ_{pp}. The contract supports the following inputs:

- INIT, giving a party a unique public key
- $(\text{SEND}, \mathsf{pk})$, sending a coin to the public key pk
- MINT, creating a new coin for the calling party
- BALANCE, returning the current balance

Transition Function Δ_{pp} (sketch)

The state transition function for a private payments system. Parties have associated public keys and balances. Parties may generate a public key, transfer and mint single-denomination coin, and query their balance.

When receiving an input $(\phi, p, \text{INIT}, \cdot, \mathsf{pk})$*:*
Assert p's public key is not set, and ensure pk is unique. Record pk as p's public key, and return it.
When receiving an input $(\phi, p, (\text{SEND}, \mathsf{pk}), \cdot, a)$*:*
If p is honest, spend from their associated public key. If not, spend from the public key a, provided it is not honestly owned. Decrease the spending key's balance by one, asserting it is non-negative. Increase pk's balance by 1.
When receiving an input $(\phi, p, \text{MINT}, \mathsf{pk}, \cdot)$*:*
Increase pk's balance by 1.
When receiving an input $(\phi, p, \text{BALANCE}, B, \cdot)$*:*
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 ledger length and the recipient – if and only if the recipient is corrupted.

When receiving an input $(\omega = (\ell, U, T, \phi), p, w)$*:*
Reject initialization transactions if ϕ is already initialized, or a transaction in U is an initializing transaction. Reject spending transactions if the coins held in ϕ, 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 ledger length ℓ, 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.

C. The Zerocash KACHINA Contract

The contract implementing Zerocash, which we will use to realize the private token contract, follows its source protocol closely, albeit with single denomination coins.

Transition Function Γ_{zc} (sketch)

The state transition function for a Zerocash 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 $(\text{SEND}, (\mathsf{pk}_z, \mathsf{pk}_e))$*:*
Process new messages through the private state oracle. Privately select an available coin to spend, retrieving its secrets. Assert that the coin's Merkle path is valid, and that the secrets are internally consistent. Compute the corresponding serial number, and publicly assert its uniqueness, marking it as spent. Publicly assert that the proven Merkle tree root is valid. Privately compute a new coin commitment and encryption, and publish these in the public state, updating the list of past Merkle roots.
When receiving an input MINT*:*
Assert the existence of secret keys. Sample a new coin commitment by the recorded private key, and privately record the commitment and associated secrets as a held coin. Add the commitment to the public set of commitments, and update the public list of past Merkle roots.
When receiving an input BALANCE*:*
Process new messages through the private state oracle. Return the size of the set of coins held in both the confirmed and projected private states.

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 $\perp$

Lemma 1. Γ_{zc} and $\mathsf{dep}_{\mathsf{zc}}$ satisfy Definition 12, and therefore the pair $(\Delta_{\mathsf{zc}}, \Lambda_{\mathsf{zc}}) := (\Delta_{\text{KACHINA}}(\Gamma_{\mathsf{zc}}), \Lambda_{\text{KACHINA}}(\Gamma_{\mathsf{zc}}, \mathsf{desc}_{\mathsf{zc}}, \mathsf{dep}_{\mathsf{zc}}))$ is in the set $\mathbb{C}_{\text{KACHINA}}$.

Proof (sketch): Transcripts generated by run-Γ fall into three categories: They set a private key (initialization), they insert a coin (minting), or they remove a coin, and insert some number of coins (sending).

Consider first a new initialization transaction. It does not affect the behavior 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 initialization transaction, as this would initialize the private keys, ensuring an abort, which violates the preconditions of dependencies.

If the new transaction is a minting or balance transaction, this 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 $\rho^{\{o,\pi\}}$, not the dynamic ρ. The only thing varying is which coins get added and removed from the set of available coins,

but this information is not directly used – its purpose is to reduce the necessary re-computation the next time around. ■

We can observe that (with some help from the simulator), the ideal Zerocash contract, given by $(\Delta_{\mathsf{zc}}, \Lambda_{\mathsf{zc}}) = (\Delta_{\text{KACHINA}}(\Gamma_{\mathsf{zc}}), \Lambda_{\text{KACHINA}}(\Gamma, \mathsf{desc}_{\mathsf{zc}}, \mathsf{dep}_{\mathsf{zc}}))$, is equivalent to the ideal private payments contract $(\Delta_{\mathsf{pp}}, \Lambda_{\mathsf{pp}})$. Formally, we instantiate two instances of $\mathcal{F}_{\mathsf{sc}}^{\Delta,\Lambda}$, as presented in Subsection III-B, and show that any attack against $(\Delta_{\mathsf{zc}}, \Lambda_{\mathsf{zc}})$ can be simulated against $(\Delta_{\mathsf{pp}}, \Lambda_{\mathsf{pp}})$.

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 [18, Appendix C.4], 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 [18, Appendix E].

Corollary 1. $\mathcal{F}_{\mathsf{sc}}^{\Delta_{\mathsf{pp}},\Lambda_{\mathsf{pp}}}$ is UC-emulated by KACHINA, parameterized 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 $\mathcal{G}_{\mathsf{simpleLedger}}$.

VI. 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 modeled 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, to construct them. This protocol utilizes non-interactive zero-knowledge proofs and state oracles to achieve the desired smart contract behavior 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 [22], and are practical to use in our setting, although they have not yet been proven in the UC model.

In ending this paper, we would like to make clear that this problem space is by no means solved. We have shown how to realize a specific class of privacy-preserving smart contracts, however privacy is not such a simple issue to be addressed by a single paper. In [18, 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, formalized, and brought into a unified model, is a crucial long-term research question for providing meaningful privacy to smart contract systems.

VII. Acknowledgements

The second and third author were partially supported by the EU Horizon 2020 project PRIVILEDGE #780477.

We thank Mikhail Volkhov and Jamie Gabbay for their feedback on early versions of this paper, and Aydin Abadi for helping to polish some of the writing.

References

[1] 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.

[2] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, 2020.

[3] 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.

[4] 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.

[5] Jan Camenisch, Manu Drijvers, and Björn Tackmann. Multi-protocol uc and its use for building modular and efficient protocols. Cryptology ePrint Archive, Report 2019/065, 2019. https://eprint.iacr.org/2019/065.

[6] 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.

[7] 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.

[8] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, 2002.

[9] 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.

[10] 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.

[11] Jacob Eberhardt and Stefan Tai. Zokrates - scalable privacy-preserving off-chain computations. In *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018*, pages 1084–1091. IEEE, 2018.

[12] The Enigma Project Team. What is Enigma? https://enigma.co/discovery-documentation/, 2019.

[13] Etherscan. Ethereum sync (default) chart. https://etherscan.io/chartsync/chaindefault, 2019.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. KACHINA – Foundations of private smart contracts (technical report). Cryptology ePrint Archive, Report 2020/543, 2020. https://eprint.iacr.org/2020/543.pdf.

[19] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. Mining for privacy: How to bootstrap a snarky blockchain. Cryptology ePrint Archive, Report 2020/401, 2020. https://eprint.iacr.org/2020/401.

[20] 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.

[21] Jonathan Lee, Kirill Nikitin, and Srinath Setty. Replicated state machines without replicated execution. Cryptology ePrint Archive, Report 2020/195, 2020. https://eprint.iacr.org/2020/195.

[22] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2111–2128. ACM Press, November 2019.

[23] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[24] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[25] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin T. Vechev. zkay: Specifying and enforcing data privacy in smart contracts. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1759–1776. ACM Press, November 2019.

[26] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[27] Fabian Vogelsteller and Vitalik Buterin. ERC-20 token standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, 2015.

[28] Jonathan Warren. Bitmessage: A peer-to-peer message authentication and delivery system. *white paper (27 November 2012), https://bitmessage. org/bitmessage. pdf*, 2012.

[29] 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.

[30] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[31] Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized Computation Platform with Guaranteed Privacy. *arXiv e-prints*, June 2015.