# Aleo VM Specification

Aleo Network Foundation

November 6, 2024

**Abstract**

We describe Aleo, a ledger-based system for decentralized private computation (DPC) that allows users to execute offline computations and produce publicly-verifiable transactions that attest to their correctness. Aleo leverages tools in the area of cryptographic proofs, including succinct *zero-knowledge proofs* and proving systems with universal and updateable structured-reference strings. The Aleo model broadly fits into the framework of DPC schemes, introduced in [BCG$^+$18a], with the notable difference being that the security definition maintains *data privacy*, but foregoes function privacy, i.e., the identity of executed functions is publicly visible, but the input and output data remains private. This paradigm allows for a more performant computational model without sacrificing essential privacy guarantees. To formalize it, we define a new notion of DPC schemes tailored for these considerations. We thoroughly document the Aleo architecture at the level of core algorithms and data structures.

## Contents

**1 Introduction** **3**
1.1 Our contributions . . . 3

**2 Techniques** **5**
2.1 Zexe to Aleo . . . 5
2.2 The Aleo records nano-kernel (RNK): an execution environment . . . 7

**3 Preliminaries** **9**
3.1 Collision-resistant hash functions . . . 9
3.2 Pseudo-random functions . . . 9
3.3 Commitment schemes . . . 9
3.4 Non-interactive zero-knowledge arguments of knowledge . . . 10
3.5 Signature schemes . . . 10

**4 Definition of Aleo DPC schemes** **12**

**5 Data structures for constructions of Aleo DPC schemes** **14**
5.1 Account . . . 14
5.2 Records . . . 14
5.3 Programs and functions . . . 15
5.4 Transitions . . . 17
5.5 Transactions . . . 18
5.6 Requests . . . 20
5.7 Ledger . . . 21

**6 Algorithms for constructions of Aleo DPC schemes** **22**
6.1 NP relations for state execution . . . 22
6.2 Virtual machines . . . 25
6.3 VM constructions for Aleo DPC schemes . . . 26

**A Appendix** **31**
A.1 Instructions . . . 31

**References** **33**

**Remark 0.1** (Disclaimer)**.** The purpose of this document is to aid the reader in understanding the ALEO snarkVM protocol. It is not meant as a one-to-one recount of the codebase[1]. There may be differences, over-simplifications, or omissions for the purposes of a smoother exposition.

## 1 Introduction

Distributed ledgers are a mechanism that maintains data across a distributed system while ensuring that every party has the same view of the data, even in the presence of corrupted parties. Ledgers can provide an indisputable history of all "events" logged in a system, thereby offering a mechanism for multiple parties to collaborate with minimal trust (any party can ensure the system's integrity by auditing history). Interest in distributed ledgers has soared recently, catalyzed by their use in cryptocurrencies (peer-to-peer payment systems) and by their potential as a foundation for new forms of financial systems, governance, and data sharing. In this work we study two limitations of ledgers, one about *privacy* and the other about *scalability*.

**A privacy problem.** The main strength of distributed ledgers is also their main weakness: the history of all transactions is available for anyone to read. This severely limits a direct application of distributed ledgers.

For example, in ledger-based payment systems such as Bitcoin [Nak09], every payment transaction reveals the payment's sender, receiver, and amount. This not only reveals private financial details of individuals and businesses using the system, but also violates fungibility, a fundamental economic property of money. This lack of privacy becomes more severe in smart contract systems like Ethereum [Woo17], wherein transactions not only contain payment details, but also embed function calls to specific applications. In these systems, every application's internal state is necessarily public, and so is the history of function calls associated to it.

This problem has motivated prior work to find ways to achieve meaningful privacy guarantees on ledgers, such as Zerocash [BCG$^+$14] and Hawk [KMS$^+$16].

**A scalability problem.** Public auditability in the aforementioned systems (and many others) is achieved via direct verification of state transitions that re-executes the associated computation. This creates the following scalability issues. First, note that in a network consisting of devices with heterogeneous computing power, requiring every node to re-execute transactions makes the weakest node a bottleneck, and this effect persists even when the underlying ledger is "perfect", that is, it confirms every valid transaction immediately. To counteract this and to discourage denial-of-service attacks whereby users send transactions that take a long time to validate, current systems introduce mechanisms such as *gas* to make users pay more for longer computations. However, such mechanisms can make it unprofitable to validate legitimate but expensive transactions, a problem known as the "Verifier's Dilemma" [LTKS15]. These problems have resulted in Bitcoin forks [Bit15] and Ethereum attacks [Eth16].

In sum, there is a dire need for techniques that facilitate the use of distributed ledgers for rich applications, without compromising privacy (of data or functions) or relying on unnecessary re-executions.

### 1.1 Our contributions

We describe ALEO (***A****utonomous* ***L****edger* ***E****xecution* ***O****ffchain*), a ledger-based system that enables users to execute offline computations and subsequently produce publicly-verifiable transactions that attest to their correctness. ALEO heavily draws inspiration from ZEXE [BCG$^+$18a] in its design and maintains its security properties, with the exception of function privacy. A system which preserves function privacy is one which obfuscates the contents (e.g., in the form of a circuit description) of the computation which is executed.

[1]https://github.com/AleoNet/snarkVM

Relaxing function privacy was a design choice taken for its considerable improvement in proof verification time and because many user-deployed applications already have large anonymity sets (Section 2.1). ALEO simultaneously supports two main security properties.

- **Data privacy:** *a transaction* consists of *state transitions*, each of which corresponds to the offline execution of a computation (arithmetized as a *circuit*). *Each state transition will reveal the identity of the computation executed, but will not reveal the identity of the participants, nor selected consumed inputs and created outputs, involved in the computation.* Furthermore, one cannot link together multiple state transitions by the same user.

- **Succinctness:** *a state transition, contained within a transaction, can be validated in time that is independent of the complexity of the offline computation whose correctness it attests to.* A transaction, an atomic unit of execution consisting of individual state transitions, incurs a verification time that is linear in the number of transitions it contains. ALEO offers a limited version of the Verifier's dilemma because the time that it takes to verify a transaction is bounded by the number of transitions it contains. The usage of *batch proving* to batch transition proofs within a single transaction into a single execution proof further amortizes the verification time.

ALEO also offers rich functionality, as offline computations in ALEO can be used to realize state transitions of multiple applications (such as digital assets, elections, and markets) simultaneously running atop the same ledger. ALEO supports this functionality by exposing a simple, yet powerful, *shared execution environment* with the following properties. For more details on the execution environment, see Section 2.2.

- **Extensibility:** users may execute arbitrary functions of their choice, though certain malicious functions may be censored based on higher level governance.
- **Isolation:** functions of malicious users cannot interfere with the computations and data of honest users.
- **Inter-process communication:** functions may exchange data with one another and invoke functions from external programs.

# 2 Techniques

In Section 2.1, we provide some of the practical (e.g., at the implementation level) shortcomings of ZEXE and address each shortcoming with the solution that ALEO provides. Next, in Section 2.2, we discuss the modified *records nano-kernel* [BCG$^+$18a] for the ALEO model.

## 2.1 ZEXE to ALEO

ZEXE offers an execution environment which aims to provide function privacy, data privacy, and succinctness of verification for its users. At a practical level, the need for function privacy drastically slows down the verification time of transactions, while data privacy already achieves the anonymity that most users seek. Additionally, ZEXE offers an execution environment where a transaction is the atomic unit for updating the state of the ledger. This increases the complexity of the model because function composition becomes scattered across independent transactions.

In this section, we discuss, in a "problem and solution" format some of the practical shortcomings in ZEXE and how ALEO addresses them.

**Problem 1: Higher complexity programs incur more network traffic.** In ZEXE, the number of transactions that correspond to the execution of a program function grows linearly with the complexity of the computation. Concretely, if a function is a composition of several other functions, relaying its execution to the network may involve submitting multiple transactions. As a toy example, consider the function composition $f \circ g$, where $f$ may be a smart contract that calls the smart contract $g$ as a subroutine. Notice that due to function privacy, an instantiation of the ZEXE protocol must globally fix the number of inputs and outputs in each transaction, so suppose we are in the 2-in, 2-out model (which is the model that ZEXE uses for its performance results).

To execute $f \circ g$, a user must submit 2 transactions—one which attests to the correctness of $f(a, b) = (c, d)$ and the other which attests to $g(s, t) = (a, b)$. Note it is impossible to fit both of these executions into a single transaction because there simply are not enough records, per transaction, in the 2-in, 2-out model to account for $a, b, c, d, s$ and $t$. With function compositions that are more complex, this effect compounds. A higher transaction count per computation incurs higher network traffic and will increase the likelihood of miscommunication (e.g., through unintended data corruption) among nodes.

One solution is to make every transaction in ZEXE $m$-in, $m$-out, for large enough $m$, to allow single transactions to capture more complex function compositions. The trade-off is that now the size of the transaction, which is linear in the number of input and output records, grows linearly in $m$. This is a poor design choice because even simple, non-complex function executions would incur high cost. The ideal behavior would be where a single transaction contains a variable amount of records, which in turn increases the expressivity of transactions (by supporting function composition).

**Solution 1: Introduce state transitions into transactions.** In comparison with the 2-in, 2-out ZEXE model, which produces 2 separate transactions for the composition $f \circ g$, the ALEO model would result only in a single transaction consisting of two state transitions, the first attesting to $f(a, b) = (c, d)$ and the second to $g(s, t) = (a, b)$. While ZEXE has a hard-coded upper-bound on the input and output records a transaction can contain, which in turn determines the fixed size of *all* transactions, ALEO supports variable storage *in the number of transitions in a transaction and in the number of input/output records each transition can contain*. As a result, the cost tradeoff between storage and expressivity per transaction that arises in ZEXE is avoided in ALEO.

Another advantage of the ALEO model is the opportunity to perform *proof batching* at the level of the transaction. Proof batching amortizes verification time because the marginal cost per proof verification decreases. For practical reasons, this can only be done among the transitions within the transaction itself, as opposed to between independent transactions, and hence is less applicable in the ZEXE model.

**Problem 2: Recursive proofs involve expensive operations.** The condition of function privacy in ZEXE imposes heavy practical constraints on the protocol, the most notable being the need for recursive SNARKs. ZEXE uses one layer of proof recursion, effectively a *proof of a proof*, to avoid leaking the verification key of the function whose execution is being verified. The *inner proof* is used for NP relations tailored to the computations of birth and death predicates. Formally, succinct proofs $\pi_{\mathsf{d}}$ *and* $\pi_{\mathsf{b}}$ *attest to the satisfaction of* $\Phi_{\mathsf{d}}$ *and* $\Phi_{\mathsf{b}}$ via the checks $\mathsf{NIZK.Verify}(\mathsf{pp}_{\Phi_{\mathsf{d}}}, \mathbb{x}_{\mathsf{e}}, \pi_{\mathsf{d}}) = 1$ and $\mathsf{NIZK.Verify}(\mathsf{pp}_{\Phi_{\mathsf{b}}}, \mathbb{x}_{\mathsf{e}}, \pi_{\mathsf{b}}) = 1$. The *outer proof* is tasked with attesting to the aforementioned verifications and is done in zero knowledge to avoid leaking the public parameters of the predicates. In more detail, this produces $\pi_{\mathsf{v}} \leftarrow \mathsf{NIZK.Prove}(\mathsf{pp}_{\Phi_{\mathsf{v}}}, \mathbb{w}_{\mathsf{v}} = (\mathsf{pp}_{\{\Phi_{\mathsf{b}},\Phi_{\mathsf{d}}\}}, \mathbb{x}_{\mathsf{e}}, \{\pi_{\mathsf{b}}, \pi_{\mathsf{d}}\}))$ to attest $\mathsf{NIZK.Verify}(\mathsf{pp}_{\{\Phi_{\mathsf{b}},\Phi_{\mathsf{d}}\}}, \mathbb{x}_{\mathsf{e}}, \{\Phi_{\mathsf{b}}, \Phi_{\mathsf{d}}\}) = 1$ allowing the verifier to check $\mathsf{NIZK.Verify}(\mathsf{pp}_{\mathsf{v}}, \mathbb{x}_{\mathsf{v}}, \pi_{\mathsf{v}}) = 1$.

The paramount challenge faced with proof recursion is costly proving times. ZEXE uses a pairing-friendly 2-chain of elliptic curves $(E_{\mathsf{BLS}}, E_{\mathsf{CP}})$ where the inner proof is produced using a zkSNARK over $E_{\mathsf{BLS}}$ and the outer proof is produced using a zkSNARK over $E_{\mathsf{CP}}$. To illustrate the need for a 2-chain of curves in the recursive setting, let $(\mathbb{F}_p, \mathbb{F}_r)$ be the prime and scalar fields for $E_{\mathsf{BLS}}$. The prover's operations occur in $\mathbb{F}_r$ while the verifier's in $\mathbb{F}_p$. Now, the prover needs to produce a proof of a correct verification, which means there is a mismatch in fields; the prover must prove, using arithmetic in $\mathbb{F}_r$, a verification which is done in $\mathbb{F}_p$. Thus, a second SNARK is needed where the prover can conduct arithmetic in $\mathbb{F}_p$. The constraint that the scalar field of the second SNARK equal $\mathbb{F}_p$ is what leads to the selection of $E_{\mathsf{CP}}$ via the Cocks-Pinch method. However, Cocks-Pinch curves are costly in time and space: in particular, the $E_{\mathsf{CP}}$ base field has roughly twice as many bits than that of $E_{\mathsf{BLS}}$ making field operations less efficient. While this issue is ameliorated in ZEXE by having the bulk of proving occur over $E_{\mathsf{BLS}}$, it remains a significant bottleneck on verification time.

**Solution 2: Relaxing function privacy.** By foregoing function privacy, it is not a requirement to avoid leaking the verification key because the identity of the function being executed is public. This removes the need for the outer SNARK over $E_{\mathsf{CP}}$ because the verification key is no longer passed in as private input to the inner SNARK. Hence the checks for predicate satisfaction $\mathsf{NIZK.Verify}(\mathsf{pp}_{\Phi_{\mathsf{d}}}, \mathbb{x}_{\mathsf{e}}, \pi_{\mathsf{d}}) = 1$ and $\mathsf{NIZK.Verify}(\mathsf{pp}_{\Phi_{\mathsf{b}}}, \mathbb{x}_{\mathsf{e}}, \pi_{\mathsf{b}}) = 1$ suffice for the privacy considerations where the circuit keys are provided in the clear. As noted above, removing the inner SNARK provides $2\times$ savings in time and space.

Moreover, this change entirely removes the needs for birth and death predicates in records because users can see the contents of the function being executed. If a record is interacting with a malicious function, this transaction can be dropped at the consensus layer. Removing predicate verification keys from records is advantageous for storage costs, but also that it makes for a cleaner construction from a programmability perspective.

**Problem 3: Inability for provers to pre-process circuit-specific keys.** Recall that a *universal* structured reference string (SRS) supports any circuit up to a given size bound by enabling anyone, in an offline phase *after* the SRS is sampled, to publicly derive a circuit-specific SRS (deterministically). In turn, the setup phase will extract the proving and verification keys from this circuit-specific SRS. ZEXE does not offer support for universal SNARKs, and hence if many users want to generate proofs for the same circuit, they have to engage in costly multi-party computation ceremonies to generate trusted randomness every time.

**Solution 3: Introduction of *deployment* objects to globally fetch circuit keys.** ALEO introduces the notion of *deployments*, allowing users to obtain circuit-specific proving and verification keys by simply querying the ledger. These circuit keys can be deterministically derived from the universal SRS and form part of the global ledger state. This model allows the user to circumvent performing an expensive trusted setup each time a new proof is generated. In the ZEXE model, such a design choice would compromise function privacy as it would leak information about the functions being executed on-chain. Note that a ZEXE-based instantiation with universal SNARKs has also been constructed in [XCZ$^+$22].

**Problem 4: Lack of on-chain computation.** Lack of on-chain computation severely limits the capabilities of ZEXE from a practical perspective because for many decentralized applications a user may want data to be publicly visible to other nodes. For example, on Ethereum, Uniswap provides a smart contract for users to exchange pairs of digital assets. The ZEXE model is extremely prohibitive for users wanting to deploy a Uniswap smart contract because all digital assets (e.g. records) are encrypted on the ledger. This is problematic in the Uniswap model because it would prevent the smart contract from updating its reserve of each asset and hence updating exchange rates.

**Solution 4: Introduce the *finalize* scope.** A hybrid model is desirable where users can produce privacy-preserving transactions but optionally have the ability to make their data, such as selected digital assets, publicly visible. ALEO provides such a feature by attributing to each program function a *finalize scope*. That is, while every function attests to the correctness in consumption and creation of private records, there are certain public inputs designated as *finalize inputs* which are used for on-chain execution. In the finalize scope, a function consumes finalize inputs to update a globally, persistent mapping on the ledger, similar to Ethereum.

Verification of the finalize scope for each transition is done by simple re-execution (there are no proofs) and occurs at the level of consensus. Roughly, for a user wanting to execute an ALEO version of the Uniswap smart contract, a state transition would contain the public, “finalize” inputs corresponding to the users’ digital assets. The execution would update the users’ accounts in a globally maintained mapping of addresses to digital assets.

## 2.2 The ALEO records nano-kernel (RNK): an execution environment

The ZEXE records-nano kernel (RNK) was constructed under the assumption of function privacy, which makes it a necessity to embed records with birth and death predicates that certify when records can be consumed or created. By relaxing function privacy, records themselves no longer need to contain descriptions of complex predicates because one can “see in the clear” whether the function is allowed to consume (or create) the record. In ALEO, we can think of state transitions as corresponding to a publicly available function, and the burden of the prover resides in showing that $f(x) = y$ for some inputs $x$ and outputs $y$. Loosely, the conditions inside of the predicates reside inside a publicly visible function. The operating system can be publicly maintained on the ledger because the logic governing transactions and state changes is publicly verifiable. By removing this layer of abstraction, we define a new variant of the RNK with the following properties:

1. **Identification**: Each record contains a unique cryptographic identifier of the program that is allowed to spend it.
2. **Proof of Transition**: Each state transition $f$ provides a succinct zero knowledge proof that attests to the correct expenditure of input records $x$ and creation of output records $y$; i.e., a proof $\pi$ that asserts $f(x) = y$.
3. **Public Validation**: Since records are bound to programs, the correctness of the execution can be publicly verified against a cryptographic proof on-chain.
4. **Local data**: While the ALEO RNK would still include transaction memorandum and auxiliary inputs, this data would not be read by predicates to authenticate whether the record can be spent. In ALEO, the concept of data ownership is more straightforward compared to ZEXE because the local data primarily consists of value-based information, such as inputs and outputs or storage of state variables, rather than predicate boolean functions that dictate record lifecycle. This simplification results in a clearer model for

data ownership.

# 3 Preliminaries

In this section, we discuss several cryptographic primitives that serve as building blocks for various protocols in this work.

## 3.1 Collision-resistant hash functions

A *collision-resistant hash function* (CRH) is a tuple of algorithms $\mathsf{CRH} = (\mathsf{Setup}, \mathsf{Eval})$ with the following syntax.

- $\mathsf{CRH.Setup}(1^\lambda) \to \mathsf{pp}_{\mathsf{CRH}}$. On input a security parameter $\lambda$ (in unary), $\mathsf{CRH.Setup}$ samples public parameters $\mathsf{pp}_{\mathsf{CRH}}$.
- $\mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, m) \to h$. On input public parameters $\mathsf{pp}_{\mathsf{CRH}}$ and a message $m$, $\mathsf{CRH.Eval}$ outputs a short hash $h$ of $m$.

**Security (informal).** Given public parameters $\mathsf{pp}_{\mathsf{CRH}} \leftarrow \mathsf{CRH.Setup}(1^\lambda)$, it is computationally infeasible to find distinct inputs $x$ and $y$ such that $\mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, x) = \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, y)$.

## 3.2 Pseudo-random functions

A *pseudo-random function* (PRF) is a tuple of algorithms $\mathsf{PRF} = (\mathsf{KeyGen}, \mathsf{Eval})$ over $\mathcal{K} \times \mathcal{X} \times \mathcal{Y}$ with the following syntax.

- $\mathsf{PRF.KeyGen}(1^\lambda) \to k$: On input a security parameter $\lambda$ (in unary), $\mathsf{PRF.KeyGen}$ outputs a key $k \in \mathcal{K}$.
- $\mathsf{PRF.Eval}(k, x) \to y$. On input a key $k \in \mathcal{K}$ and a message $x \in \mathcal{X}$, $\mathsf{PRF.Eval}$ outputs a message $y \in \mathcal{Y}$.

**Security (informal).** $\mathsf{PRF}$ is computationally indistinguishable from a random function over $\mathcal{K} \times \mathcal{X} \times \mathcal{Y}$.

## 3.3 Commitment schemes

A *commitment scheme* (CM) is a tuple of algorithms $\mathsf{CM} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ with the following syntax.

- $\mathsf{CM.Setup}(\lambda) \to \mathsf{pp}_{\mathsf{CM}}$. On input a security parameter $\lambda$ (in unary), $\mathsf{CM.Setup}$ samples public parameters $\mathsf{pp}_{\mathsf{CM}}$.
- $\mathsf{CM.Commit}(\mathsf{pp}_{\mathsf{CM}}, m, r) \to \mathsf{cm}$. On input public parameters $\mathsf{pp}_{\mathsf{CM}}$, a message $m$, and optional hiding randomness $r$, $\mathsf{CM.Commit}$ outputs a commitment $\mathsf{cm}$ to $m$.
- $\mathsf{CM.Open}(\mathsf{pp}_{\mathsf{CM}}, m, r) \to o$. On input public parameters $\mathsf{pp}_{\mathsf{CM}}$, a message $m$ and optional hiding randomness $r$, $\mathsf{CM.Open}$ outputs an opening $o$.
- $\mathsf{CM.Verify}(\mathsf{pp}_{\mathsf{CM}}, \mathsf{cm}, o) \to b$. On input public parameters $\mathsf{pp}_{\mathsf{CM}}$, a commitment $\mathsf{cm}$, and an opening $o$, $\mathsf{CM.Verify}$ outputs a bit $b$ indicating that the opening $o$ opens the commitment $\mathsf{cm}$ to the message $m$.

**Security (informal).** CM schemes are considered secure if they satisfy the following two properties.

- **Binding.** Given public parameters $\mathsf{pp}_{\mathsf{CM}} \leftarrow \mathsf{CM.Setup}(1^\lambda)$, CM is computationally binding if no efficient adversary can produce a commitment $\mathsf{cm}$ and openings $o, o'$ that open $\mathsf{cm}$ to distinct messages $m, m'$.
- **Hiding.** Given public parameters $\mathsf{pp}_{\mathsf{CM}} \leftarrow \mathsf{CM.Setup}(1^\lambda)$ and any pair of messages $m, m'$, CM is computationally hiding if no efficient adversary can distinguish between the distributions of $\mathsf{CM.Commit}$ when committing to $m$ and of $\mathsf{CM.Commit}$ when committing to $m'$.

### 3.4 Non-interactive zero-knowledge arguments of knowledge

*Non-interactive zero knowledge arguments of knowledge* (NIZK) enable a party, known as the *prover*, to convince another party, known as the *verifier*, about knowledge of the witness for an $\mathsf{NP}$ statement without revealing any information about the witness (besides what is already implied by the statement being true). In this work, we consider special types of NIZKs with universal setup, which means that a singular, "universal" set of public parameters can be used to derive relation-specific circuit keys. Formally, a NIZK with universal setup is a tuple $\mathsf{NIZK} = (\mathsf{UnivSetup}, \mathsf{Specialize}, \mathsf{Prove}, \mathsf{Verify})$ with the following syntax.

- $\mathsf{NIZK}.\mathsf{UnivSetup}(1^\lambda) \to \mathsf{pp}_\mathsf{U}$. On input a security parameter $\lambda$ (in unary), $\mathsf{NIZK}.\mathsf{Setup}$ outputs a set of universal public parameters $\mathsf{pp}_\mathsf{U}$ (also known as a *common reference string*).
- $\mathsf{NIZK}.\mathsf{Specialize}(\mathsf{pp}_\mathsf{U}, \mathcal{R}) \to (\mathsf{ipk}, \mathsf{ivk})$. On input a set of universal parameters $\mathsf{pp}_\mathsf{U}$ and the specification of an $\mathsf{NP}$ relation $\mathcal{R}$, $\mathsf{NIZK}.\mathsf{Specialize}$ deterministically specializes a proving key $\mathsf{ipk}$ and verifying key $\mathsf{ivk}$.
- $\mathsf{NIZK}.\mathsf{Prove}(\mathsf{ipk}, \mathbb{x}, \mathbb{w})$. On input a proving key $\mathsf{ipk}$, an instance-witness pair $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, $\mathsf{NIZK}.\mathsf{Prove}$ outputs a proof $\pi$.
- $\mathsf{NIZK}.\mathsf{Verify}(\mathsf{ivk}, \mathbb{x}, \pi) \to b$. On input a verifying key $\mathsf{ivk}$, an instance $\mathbb{x}$, and a proof $\pi$, $\mathsf{NIZK}.\mathsf{Verify}$ outputs a decision bit $b$.

**Security (informal).** $\mathsf{NIZK}$ schemes with universal setup satisfy the following security properties.

- **Completeness**. Honestly generated proofs always make the verifier accept.
- **Knowledge-soundness**. For every efficient adversary that makes the verifier accept, there exists an efficient extractor that can "extract" a witness for the relation $\mathcal{R}$ by running the adversary internally.
- **Perfect zero-knoweldge**. states that honestly generated proofs can be perfectly simulated, when given a trapdoor to the public parameters. In fact, we require a strong form of (computational) proof of knowledge known as *simulation-extractability*, which states that proofs continue to be proofs of knowledge even when the adversary has seen prior simulated proofs.

### 3.5 Signature schemes

A *signature scheme* is a tuple of algorithms $\mathsf{SIG} = (\mathsf{Setup}, \mathsf{Keygen}, \mathsf{Sign}, \mathsf{Verify})$ with the following syntax.

- $\mathsf{SIG}.\mathsf{Setup}(1^\lambda) \to \mathsf{pp}_\mathsf{SIG}$. On input a security parameter $\lambda$ (in unary), $\mathsf{SIG}.\mathsf{Setup}$ samples public parameters $\mathsf{pp}_\mathsf{SIG}$.
- $\mathsf{SIG}.\mathsf{Keygen}(\mathsf{pp}_\mathsf{SIG})$. On input public parameters $\mathsf{pp}_\mathsf{SIG}$, $\mathsf{SIG}.\mathsf{Keygen}$ samples a key pair $(\mathsf{pk}_\mathsf{SIG}, \mathsf{sk}_\mathsf{SIG})$.
- $\mathsf{SIG}.\mathsf{Sign}(\mathsf{pp}_\mathsf{SIG}, \mathsf{sk}_\mathsf{SIG}, m, r) \to \sigma$. On input public parameters $\mathsf{pp}_\mathsf{SIG}$, secret key $\mathsf{sk}_\mathsf{SIG}$, message $m$, and challenge randomness $r$, $\mathsf{SIG}.\mathsf{Sign}$ produces a signature $\sigma$.
- $\mathsf{SIG}.\mathsf{Verify}(\mathsf{pp}_\mathsf{SIG}, \mathsf{pk}_\mathsf{SIG}, m, \sigma) \to b$. On input public parameters $\mathsf{pp}_\mathsf{SIG}$, public key $\mathsf{pk}_\mathsf{SIG}$, message $m$, and signature $\sigma$, $\mathsf{SIG}.\mathsf{Verify}$ outputs a bit $b$ denoting whether $\sigma$ is a valid signature for $m$ under public key $\mathsf{pk}_\mathsf{SIG}$.

**Security (informal).** $\mathsf{SIG}$ satisfies *existential unforgeability*, defined as follows. Given a public key $\mathsf{pk}_\mathsf{SIG}$, it is infeasible to produce a forgery under $\mathsf{pk}_\mathsf{SIG}$ or *under under any randomization of* $\mathsf{pk}_\mathsf{SIG}$. This notion strengthens the standard unforgeability notion, and is similar to that of randomizable signatures in [BS23].

#### 3.5.1 Randomizable signatures

A *randomizable* signature scheme is a tuple of algorithms $\mathsf{SIG} = (\mathsf{Setup}, \mathsf{Keygen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{RandPk}, \mathsf{RandSig})$ that enables a party to sign messages, while also allowing randomization of public keys and signatures to prevent linking across multiple signatures. In addition to the usual algorithms above, $\mathsf{SIG}$ has two algorithms for randomizing public keys and signatures.

- $\mathsf{SIG.RandPk}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{SIG}}, r_{\mathsf{SIG}})$. On input public parameters $\mathsf{pp}_{\mathsf{SIG}}$ and public key $\mathsf{pk}_{\mathsf{SIG}}$, $\mathsf{SIG.RandPk}$ samples a randomized public key $\hat{\mathsf{pk}}_{\mathsf{SIG}}$.
- $\mathsf{SIG.RandSig}(\mathsf{pp}_{\mathsf{SIG}}, \sigma, r_{\mathsf{SIG}}) \to \hat{\sigma}$. On input public parameters $\mathsf{pp}_{\mathsf{SIG}}$ and a signature $\sigma$, $\mathsf{SIG.RandSig}$ samples a randomized signature $\hat{\sigma}$.

**Security (informal).** $\mathsf{SIG}$ must satisfy the following security properties.

- *Existential unforgeability.* Given a public key $\mathsf{pk}_{\mathsf{SIG}}$, it is infeasible to produce a forgery under $\mathsf{pk}_{\mathsf{SIG}}$ or *under under any randomization of* $\mathsf{pk}_{\mathsf{SIG}}$. This notion strengthens the standard unforgeability notion, and is similar to that of randomizable signatures in [BS23].
- *Unlinkability.* Given a public key $\mathsf{pk}_{\mathsf{SIG}}$ and a tuple $(\hat{\mathsf{pk}}_{\mathsf{SIG}}, m, \hat{\sigma})$ where $\hat{\sigma}$ is a valid signature for $m$ under $\hat{\mathsf{pk}}_{\mathsf{SIG}}$, no efficient adversary can determine if $\hat{\mathsf{pk}}_{\mathsf{SIG}}$ is a fresh public key and $\hat{\sigma}$ a fresh signature, or if instead $\hat{\mathsf{pk}}_{\mathsf{SIG}}$ is a randomization of $\mathsf{pk}_{\mathsf{SIG}}$ and $\hat{\sigma}$ a randomization of a signature for $\mathsf{pk}_{\mathsf{SIG}}$. This property is a computational relaxation of the perfect unlinkability property of randomizable signatures in [BS23].
- *Injective randomization.* Randomization of public keys is *(computationally) injective* with respect to randomness. Informally, given public parameters $\mathsf{pp}_{\mathsf{SIG}}$, it is infeasible to find a public key $\mathsf{pk}_{\mathsf{SIG}}$ and $r_1 \neq r_2$ such that $\mathsf{SIG.RandPk}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{SIG}}, r_1) = \mathsf{SIG.RandPk}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{SIG}}, r_2)$.

## 4 Definition of ALEO DPC schemes

Bowe et. al. [BCG+18b] define *decentralized private computation* (DPC) schemes, a cryptographic primitive in which parties with access to an ideal append-only ledger execute computations offline and subsequently post privacy-preserving, publicly-verifiable transactions that attest to the correctness of these offline executions.

In this section, we define the ALEO DPC scheme, denoted by $\mathsf{DPC}_{\mathrm{A}}$, as an abstract cryptographic primitive. $\mathsf{DPC}_{\mathrm{A}}$ differs from the traditional DPC schemes of [BCG+18b] in the following ways:

- *The removal of function privacy.*
- *The ability to perform multiple function executions within a single state change.*
- *The ability to perform on-chain execution.*
- *The ability to synthesize circuit-specific keys deterministically provided universal parameters.*

In $\mathsf{DPC}_{\mathrm{A}}$, there are five fundamental data structures: *accounts*, *records*, *transitions*, *transactions*, and the *ledger*, discussed in detail in Section 5. Formally, $\mathsf{DPC}_{\mathrm{A}}$ is defined as the following tuple:

$$\mathsf{DPC}_{\mathrm{A}} = (\mathsf{Setup}, \mathsf{Authorize}, \mathsf{ExecuteTr}, \mathsf{ExecuteTx}, \mathsf{Finalize}, \mathsf{Synthesize}, \mathsf{Verify})$$

The syntax and semantics of these algorithms are informally described below.

**Setup:** $\mathsf{DPC}_{\mathrm{ALEO}}.\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{ask}, \mathsf{ack}, \mathsf{avk}, \mathsf{apk})$.

On input a security parameter $1^\lambda$, $\mathsf{DPC.Setup}$ outputs public parameters $\mathsf{pp}$ for the system, along with the following account keys: the account private key $\mathsf{ask}$, the account compute key $\mathsf{ack}$, the account view key $\mathsf{avk}$, and the account public key $\mathsf{apk}$. A trusted party is responsible for generating the public parameters.

For some constructions, the trusted party can be replaced by an efficient multiparty computation that securely realizes the $\mathsf{DPC.Setup}$ algorithm (see [BCG+15, ZCa16, BGM17, BGG18] for how this has been done in some systems); in other constructions, the trusted party may not be needed, as the public parameters may simply consist of a random string of a certain length.

**Execute transition.** Any user invokes $\mathsf{DPC}_{\mathrm{A}}.\mathsf{ExecuteTr}$ to consume records and create new ones, and create state transitions for the ledger.

$$\mathsf{ExecuteTr}^{\mathbf{L}} \begin{pmatrix} \text{old records} & [\mathbf{r}_i]_1^m \\ \text{old record serial numbers} & [\mathsf{sn}_i]_1^m \\ \text{old record tags} & [\mathsf{tag}_i]_1^m \\ \text{new record payloads} & [\mathsf{payload}_j]_1^n \\ \text{new address compute keys} & [\mathsf{ck}_j]_1^n \\ \text{new address public keys} & [\mathsf{apk}_j]_1^n \\ \text{execution function} & f \end{pmatrix} \to \begin{pmatrix} \text{new records} & [\mathbf{r}_j]_1^n \\ \text{new record serial numbers} & [\mathsf{sn}_j]_1^n \\ \text{new record record tags} & [\mathsf{tag}_j]_1^n \\ \text{new record record commitments} & [\mathsf{cm}_j]_1^n \\ \text{transition} & \mathsf{t} \end{pmatrix}.$$

On input a list of old records $[\mathbf{r}_i]_1^m$ and their serial numbers, attributes for new records, the execution function $f$, and other account specific information, $\mathsf{DPC}_{\mathrm{A}}.\mathsf{ExecuteTr}$ produces new records $[\mathbf{r}_j]_1^n$, along with their commitments, and a transition $\mathsf{t}$.

**Execute transaction.** Any user may invoke $\mathsf{DPC.ExecuteTx}$ to composte a transaction out of transitions.

$$\mathsf{ExecuteTx}^{\mathbf{L}} \begin{pmatrix} \text{old records for } [t_k]_1^{n_{\mathsf{Transitions}}} & [\{[\mathbf{r}_i]_1^m\}_k]_1^{n_{\mathsf{Transitions}}} \\ \text{new records for } [t_k]_1^{n_{\mathsf{Transitions}}} & [\{[\mathbf{r}_j]_1^n\}_k]_1^{n_{\mathsf{Transitions}}} \\ \text{collection of transitions} & [t_k]_1^{n_{\mathsf{Transitions}}} \end{pmatrix} \to \begin{pmatrix} \text{transaction} & \mathsf{tx} \\ \mathcal{R}_{\mathsf{ex}} \text{ instance} & \mathbb{x}_{\mathsf{ex}} \\ \text{execution proof} & \pi \end{pmatrix}.$$

On input a list of transitions along with the records being consumed and created for each transition, $\mathsf{DPC}_{\mathrm{A}}.\mathsf{ExecuteTx}$ outputs a transaction $\mathsf{tx}$, a public instance $\mathbb{x}_{\mathsf{ex}}$ for the execute relation $\mathcal{R}_{\mathsf{ex}}$, and a proof

$\pi$ for $\mathcal{R}_{\mathsf{ex}}$. The proof $\pi$ attests that for each transition $\mathtt{t}$, the consumption of input records by $f$ correctly produced the output records, and that the new record attributes were correctly derived from the inputs. For a formal definition of $\mathcal{R}_{\mathsf{ex}}$, see Definition 6.5. The user subsequently pushes $\mathsf{tx}$ to the ledger by invoking $\mathbf{L}.\mathsf{Push}(\mathsf{tx})$.

**Finalize:** $\mathsf{DPC}_{\mathrm{A}}.\mathsf{Finalize}(\mathsf{addr}, v) \to b$.

On input an address $\mathsf{addr}$ and a value $v$, $\mathsf{DPC}_{\mathrm{A}}.\mathsf{Finalize}$ updates the global persistent mapping in the ledger $\mathbf{L}$ keyed at $\mathsf{addr}$ to contain the value $v$, and outputs a bit $b$ indicating the success or failure of the operation.

**Synthesize:** $\mathsf{DPC}_{\mathrm{A}}.\mathsf{Synthesize}(\mathsf{pp}, \mathcal{R}) \to (\mathsf{ipk}, \mathsf{ivk})$.

On input the public parameters $\mathsf{pp}$ and a $\mathsf{NP}$ relation $\mathcal{R}$, $\mathsf{DPC}_{\mathrm{A}}.\mathsf{Synthesize}$ synthesizes circuit-specific keys deterministically by invoking $\mathsf{NIZK}.\mathsf{Specialize}(\mathsf{pp}, \mathcal{R})$.

**Verify:** $\mathsf{DPC}_{\mathrm{A}}.\mathsf{Verify}^{\mathbf{L}}(\mathsf{pp}, \mathsf{tx}) \to b$.

On input public parameters $\mathsf{pp}$ and a transaction $\mathsf{tx}$, and given oracle access to the ledger $\mathbf{L}$, $\mathsf{DPC}.\mathsf{Verify}$ outputs a bit $b$ denoting whether the transaction $\mathsf{tx}$ is valid relative to the ledger $\mathbf{L}$.

# 5 Data structures for constructions of ALEO DPC schemes

## 5.1 Account

An *account*, denoted by the symbol $\mathcal{A}$, is a data structure that contains cryptographic keys to encrypt or decrypt records (Section 5.2) and sign requests (Section 5.6). Formally, an account is a tuple $\mathcal{A} = (\mathsf{ask}, \mathsf{ack}, \mathsf{avk}, \mathsf{apk})$ with the following syntax.

- $\mathsf{ask} = (\mathsf{sk}_{\mathsf{SIG}}, r_{\mathsf{SIG}})$ is the account private key, which consists of the following components:
  - $\mathsf{sk}_{\mathsf{SIG}}$ is the account signing key that is sampled from $(\mathsf{sk}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{SIG}}) \leftarrow \mathsf{SIG.Keygen}(\mathsf{pp}_{\mathsf{SIG}})$.
  - $r_{\mathsf{SIG}}$ is the signature randomizer randomly sampled from $\mathbb{F}$.
- $\mathsf{ack} = (\mathsf{pk}_{\mathsf{SIG}}, \mathsf{pr}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{PRF}})$ is the account compute key, which consists of the following components:
  - $\mathsf{pk}_{\mathsf{SIG}}$ is the account public key, as sampled above, and is a commitment to $\mathsf{sk}_{\mathsf{SIG}}$.
  - $\mathsf{pr}_{\mathsf{SIG}}$ is a commitment to $r_{\mathsf{SIG}}$.
  - $\mathsf{sk}_{\mathsf{PRF}}$ is the $\mathsf{PRF}$ private key and is computed as $\mathsf{sk}_{\mathsf{PRF}} := \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{pk}_{\mathsf{SIG}} || \mathsf{pr}_{\mathsf{SIG}})$.
- $\mathsf{avk}$ is the account view key, derived from $\mathsf{ask}$, and is computed as $\mathsf{avk} := \mathsf{sk}_{\mathsf{SIG}} + r_{\mathsf{SIG}} + \mathsf{sk}_{\mathsf{PRF}}$.
- $\mathsf{apk}$ is the account address and is computed as $\mathsf{apk} := \mathsf{pk}_{\mathsf{SIG}} + \mathsf{pr}_{\mathsf{SIG}} + \mathsf{pk}_{\mathsf{PRF}}$, where $\mathsf{pk}_{\mathsf{PRF}}$ is a commitment to $\mathsf{sk}_{\mathsf{PRF}}$.

## 5.2 Records

A *record*, denoted by the symbol $\mathbf{r}$, is a data structure that represents ownership over some unit of data. Records can be created or consumed to update the global state of the distributed protocol. A record is *created* when its commitment $\mathsf{cm}$ is posted to the ledger as part of a transaction, and *consumed* when its serial number, or *nullifier*, $\mathsf{sn}$ appears on the ledger as part of a later transaction.

To consume a record, a user must be able to prove that they know the account $\mathsf{PRF}$ private key $\mathsf{sk}_{\mathsf{PRF}}$ from which serial numbers are derived and the unique record nonce $\rho$ assigned to the record. The ledger forbids the same serial number to appear more than once, so that: (a) a record cannot be consumed twice, or ‘double-spent”, because it is in unique correspondence with its serial number; (b) others cannot prevent one from consuming a record because it is computationally infeasible to create two distinct records that share the same serial number but have distinct commitments.

Formally, a *record* is a tuple $\mathbf{r} = (v, \mathsf{pid}, \mathsf{apk}, \mathsf{d}, \rho, r)$ where:

- $v$: the visibility mode of the record, which can be either public or private.
- $\mathsf{pid}$: the program id of the program which owns the record.
- $\mathsf{apk}$: the address public key of the record owner, derived from the account private key $\mathsf{ask}$ (Section 5.1).
- $\mathsf{d}$: a data payload containing arbitrary application-dependent information.
- $\rho$: a nonce, which is used to produce the unique serial number of a record. (Section 5.2).
- $r$: additional randomness used to make the record commitment hiding.

**Serial Number.** A *serial number*, or nullifier, is a unique identifier for a record $\mathbf{r}$ which when posted onto the ledger indicates that the associated record has been spent. Given a public pseudo-random function $\mathsf{PRF}$, the serial number $\mathsf{sn}$ is computed as

$$\mathsf{sn} := \mathsf{PRF}(\mathsf{sk}_{\mathsf{PRF}}, \rho)$$

where $\mathsf{sk}_{\mathsf{PRF}}$ is retrieved from the account compute key (Section 5.1) of $\mathbf{r}$'s owner address, and $\rho$ is the record nonce of $\mathbf{r}$.

**Record Commitment.** A *record commitment* is a binding and hiding commitment to $\mathbf{r} := (v, \mathsf{pid}, \mathsf{apk}, \mathsf{d}, \rho, r)$ and is computed as:

$$\mathsf{cm} := \mathsf{CM.Commit}(\mathsf{pp}_{\mathsf{CM}}, v||\mathsf{apk}||\mathsf{d}||\rho; r)$$

where $r$ is the record hiding randomness.

**Record Tag.** A *record tag*, denoted as $\mathsf{tag}$, is used to keep track of consumed records, as is computed via:

$$\mathsf{tag} := \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{gk}||\mathsf{cm}||r)$$

where $\mathsf{gk} \leftarrow \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{avk}||r)$ is called the *graph key*, and $r$ is record hiding randomness, $\mathsf{cm}$ is the record commitment, and $\mathsf{avk}$ is the account view key.

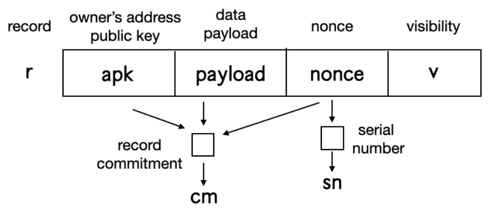


**Figure 1:** Diagram of a record.

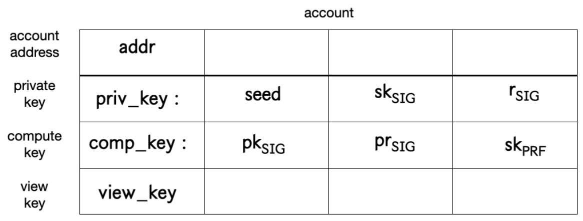


**Figure 2:** Diagram of an account.

## 5.3 Programs and functions

### 5.3.1 Programs

A *program*, denoted by $\mathcal{P}$, acts as a container to group user-specified collections of functions. Formally, a program is a tuple $(\mathsf{pid}, \boldsymbol{f}, \mathsf{d})$ where:

- $\mathsf{pid}$ is a unique identifier for $\mathcal{P}$.
- $\boldsymbol{f}$ is a list of functions contained within the scope of $\mathcal{P}$ (Section 5.3.2).
- $\mathsf{d}$ is a data payload which contains arbitrary program-specific information, such as metadata about the types of records being consumed or created.

### 5.3.2 Functions

A *function*, denoted by $f$, exists within the scope of a program (Section 5.3.1) and contains a sequential set of instructions that perform a specific task. A function contains the necessary information to create state transitions (Section 5.4), which are atomic units of programmable execution. Formally, an $n$-in, $m$-out function $f$ is a tuple $(\mathsf{pid}, \mathsf{fid}, [\mathsf{ti}_j]_1^n, [\mathsf{to}_j]_1^m, \boldsymbol{\mathcal{I}}, \boldsymbol{\mathcal{F}})$ where:

- $\mathsf{pid}$ is the program ID of the program that contains.
- $\mathsf{fid}$ is the function ID is a unique identifier for $f$ computed as $\mathsf{fid} := \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{pid}||\mathsf{fid})$.
- $[\mathsf{ti}_j]_1^n$ is a list of $n$ function input types which can be either of the type public plaintext, private plaintext, or record.

- $[\mathsf{to}_j]_1^m$ is a list of $m$ function output types which can be either of the type public plaintext, private plaintext, or record.
- $\mathcal{I} = \{\mathcal{I}_k\}_1^s$ is a sequential list of ALEO instructions that define the programmable functionality of $f$. For the currently supported set of instructions, see Appendix A.1.
- $\mathcal{F} = (\mathcal{F}_I, \mathcal{F}_C)$ is the finalize scope, which consists of a set of finalize inputs $\mathcal{F}_I$ and an ordered list of finalize commands $\mathcal{F}_C$.

**Non-record function arguments.** Functions can take as input and output unowned units of data, or *non-record arguments*. Unlike a record, these units of data do not have owners associated to their consumption or creation. A non-record argument $\mathsf{x}$ is a tuple $(v, \mathsf{d})$ where $v$ is a visibility mode that is either public or private, and $\mathsf{d}$ is a data payload. The payload $\mathsf{d}$ is either a plaintext or ciphertext depending on its visibility.

**Finalize scope.** The *finalize scope* is an abstraction to support on-chain execution. The ledger $\mathbf{L}$ contains globally persistent mappings from addresses to values that can be updated by users via $\mathbf{L}.\mathsf{UpdateMappings}$. To execute a function in the finalize scope $\mathcal{F} = (\mathcal{F}_I, \mathcal{F}_C)$, a user executes the commands $\mathcal{F}_C$ which act on the inputs $\mathcal{F}_I$.

**NP relation for function satisfaction.** Every program function can be thought of as an *arithmetization* of an NP relation $\mathcal{R}_f$; more precisely, an instance-witness tuple $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}_f$ if and only if $f(\mathbb{x}, \mathbb{w}) = 0$. In the context of having records as inputs and outputs and public or private inputs, we formalize the function satisfaction relation below in Definition 5.1.

**Definition 5.1** (Function satisfaction relation)**.** *The* NP *relation $\mathcal{R}_f$ is the set of all tuples* $(\mathbb{x}, \mathbb{w})$

$$\mathbb{x} = \left( \begin{array}{ll} \textit{program and function} & (\mathcal{P}, f) \\ \textit{serial numbers of input records} & [\mathsf{sn}_i]_1^n \\ \textit{commitments of output records} & [\mathsf{cm}'_i]_1^m \\ \textit{input IDs of non-records} & [\mathsf{id}_i]_1^s \\ \textit{output IDs of non-records} & [\mathsf{id}'_i]_1^k \\ \textit{public non-record inputs} & [\mathsf{i}_j]_1^n \end{array} \right), \mathbb{w} = \left( \begin{array}{ll} \textit{account compute key} & \mathsf{ack} \\ \textit{input records} & [\mathbf{r}_i]_1^m \\ \textit{output records} & [\mathbf{r}'_j]_1^n \\ \textit{private non-record inputs} & [\mathsf{i}_j]_1^n \\ \textit{private non-record outputs} & [\mathsf{o}_j]_1^n \\ \textit{transition view key} & \mathsf{tvk} \\ \textit{input view keys} & [\mathsf{ivk}_j]_1^n \end{array} \right)$$

*that satisfy the following conditions.*

$\mathcal{R}_f(\mathbb{x}, \mathbb{w})$ :

1. For the input records, do as follows. For each $i \in \{1, \ldots, n\}$:
   - Parse the input record $\mathbf{r}_i$ as $(v, \mathsf{pid}, \mathsf{apk}, \mathsf{d}, \rho, r)$.
   - Check the serial number $\mathsf{sn}_i$ is valid: $\mathsf{sn}_i := \mathsf{PRF}(\mathsf{sk}_{\mathsf{PRF}}, \rho)$.
2. For the input non-records, do as follows. For each $j \in \{1, \ldots, s\}$:
   - Parse the non-record input $\mathsf{i}_j$ as $(v_j, \mathsf{d}_j)$.
   - If $v_j$ is public, check the input id $c_j$ is valid: $c_j := \mathsf{CRH.Eval}(\mathsf{fid}, \mathsf{tvk}, j, \mathsf{d}_j)$.
   - If $v_j$ is private, check the input id $c_j$ is valid: $\mathsf{d}_j = \mathsf{ENC.Dec}(\mathsf{pp}_{\mathsf{ENC}}, \mathsf{ivk}_j, c_j)$.
3. For the output records, do as follows. For each $i \in \{1, \ldots m\}$:
   - Parse the input output $\mathbf{r}'_i$ as $(v, \mathsf{pid}, \mathsf{apk}, \mathsf{d}, \rho, r)$.
   - Check the record commitment $\mathsf{cm}_i$ is valid: $\mathsf{cm}_i := \mathsf{CM.Commit}(\mathsf{pp}_{\mathsf{CM}}, v||\mathsf{apk}||\mathsf{d}||\rho; r)$.
4. For the output non-records, do as follows. For each $j \in \{1, \ldots, k\}$:
   - Parse the non-record output $\mathsf{o}_j$ as $(v_j, \mathsf{d}_j)$.

- If $v_j$ is public, check the output id $c_j$ is valid: $c_j := \mathsf{CRH.Eval}(\mathsf{fid}, \mathsf{tvk}, j, \mathsf{d}_j)$.
- If $v_j$ is private, check the output id $c_j$ is valid: $\mathsf{d}_j = \mathsf{ENC.Dec}(\mathsf{pp}_{\mathsf{ENC}}, \mathsf{ivk}_j, c_j)$.

5. Check the function evaluation: $f([\mathbf{r}_i]_1^n, [\mathsf{i}_i]_1^n) = ([\mathbf{r}'_i]_1^n, [\mathsf{o}_i]_1^n)$.

## 5.4 Transitions

A *transition* corresponds to the atomic execution of a singular program function. Batches of transitions, whose associated functions are all in the scope of the same program, comprise an execution transaction (Section 5.5.1). In the *unbatched* construction, a transition contains a zero-knowledge proof attesting to the correctness of the (offline) computation of the executed program function; in particular, for a function $f$, the proof attests that $f$, when evaluated on the set of specified inputs, produces the set of claimed outputs. In the *batched* construction, there is a single proof that attests to the correctness of all the transitions within the transaction, and is thus placed at the level of the transaction. In this section, we describe the batched transition construction.

In order to produce a transition for off-chain execution, a user selects some previously-created records to consume, assembles new records to create, and decides on other aspects of local data such as auxiliary inputs. Then, the user will locally evaluate the function which the transition corresponds to, which may lead to the creation of additional function executions depending on the call diagram of the parent function. In our construction, we restrict program functions to only be able to invoke *external* program functions to avoid recursive pathologies. Note a transition will reveal the identity of the program function which is executed and the number of input records consumed and output records created[2].

Each transition is associated with a transition view key, denoted by $\mathsf{tvk}$, derived from the account view key $\mathsf{avk}$, which is a cryptographic key that allows the user to encrypt the contents of private non-record inputs and outputs. Recall that the payload and address of records are encrypted using the $\mathsf{avk}$ itself, rather than the $\mathsf{tvk}$. From the $\mathsf{tvk}$, the user can derive the transition commitment, denoted by $\mathsf{tcm}$, as a commitment to the $\mathsf{tvk}$. Formally, a transition $T$ that corresponds to the execution of an $n$-in, $m$-out function $f$ is a tuple $(\mathsf{tid}, \mathsf{pid}, \mathsf{fid}, [\mathsf{i}_j]_1^n, [\mathsf{o}_j]_1^m, \mathsf{tpk}, \mathsf{tcm}, \mathsf{scm})$ where:

- $\mathsf{tid}$: the transition id, a unique identifier for the transition, computed as the hash of the input and outputs commitments *of records* via $\mathsf{tid} := \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{cm}_{i_1} || \ldots, \mathsf{cm}_{i_s} || \mathsf{cm}'_{j_1} || \ldots || \mathsf{cm}'_{j_d} || \mathsf{tcm})$. Note this means that even if a function contains non-record inputs, they will not influence the resulting transition id. The transition commitment $\mathsf{tcm}$, defined below, is appended to the pre-image to ensure that 0-in, 0-out transitions have unique transition ids.
- $\mathsf{pid}$: the program id of the program containing the function which the transition corresponds to.
- $\mathsf{fid}$: the function id of the function which the transition corresponds to.
- $[\mathsf{i}_j]_1^n$: a list of $n$ inputs to $f$ which can be either of the type public plaintext, private plaintext, or record.
- $[\mathsf{o}_j]_1^m$: a list of $m$ outputs of $f$ which can be either of the type public plaintext, private plaintext, or record.
- $\mathsf{tcm}$: A commitment to the transition view key: $\mathsf{tcm} := \mathsf{CM.Commit}(\mathsf{pp}_{\mathsf{CM}}, \mathsf{tvk}; r)$, where $r$ is hiding randomness. In Section 5.6, $r$ corresponds to the transition signing key $\mathsf{tsk}$.
- $\mathsf{tpk}$: is the transition public key. In the context of delegating proving to a trusted-third party, $\mathsf{tpk}$ ensures that the *request* (Section 5.6) to execute a transition comes from the "transition owner", i.e., the owner of the records. The transition public key is computed by sampling $(\mathsf{tpk}, \mathsf{tsk}) \leftarrow \mathsf{SIG.Setup}(1^\lambda)$.

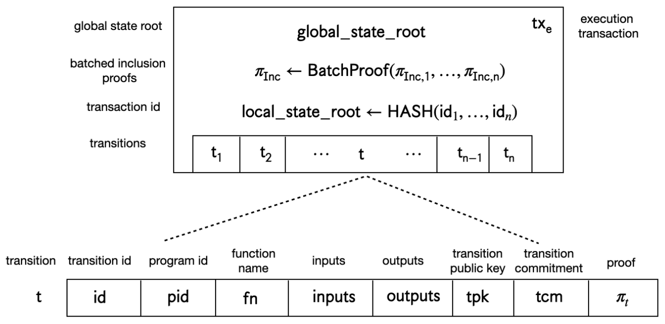
[2]However, the number of input and output records in a transition can be masked through the use of *dummy records*.

## 5.5 Transactions

A *transaction* can be of two different kinds: an *execution transaction* or a *deployment transaction*. An execution transaction, sometimes referred to as just an *execution*, consists of multiple transitions, each corresponding to the excution of a singular program function. The transitions in an execution can be viewed as a stack of function executions, where the top-most element contains the root, or parent, process while the remainder are the child processes. Additionally, an execution consists of a singular zero-knowledge proof which attests to the correct function execution of the program functions inside the transitions and the correct consumption and creation of input and output records. For a more granular analysis on the $\mathsf{NP}$ relations which an execution transaction verifies, see Section 6.1.

A deployment transaction, or just a *deployment*, consists of a program along with the circuit keys for a zero-knowledge, non-interactive argument of knowledge ($\mathsf{NIZK}$) for each individual function execution. More precisely, for a function $f \in \mathcal{P}$, these are the circuit keys for a $\mathsf{NIZK}$ argument for the relation $\mathcal{R}_f$ as defined in Section 5.3.2. These circuit keys consist of a public proving key $\mathsf{ipk}$ and verifying key $\mathsf{ivk}$ that allow a prover to compute succinct proofs and any verifier to efficiently verify them. In contrast to ZEXE, the $\mathsf{ipk}$ and $\mathsf{ivk}$ can be public keys given the design choice to forego function privacy. This removes the extra layer of proof recursion [BGG18] that was used in ZEXE since the $\mathsf{ipk}$ and $\mathsf{ivk}$ can be used in the clear.

In Section 5.5.1 we describe the execution data structure and in Section 5.5.2 the deployment data structure.



global state root
global_state_root
$\mathsf{tx_e}$
execution transaction
batched inclusion proofs
$\pi_{\mathsf{Inc}} \leftarrow \mathsf{BatchProof}(\pi_{\mathsf{Inc},1}, \ldots, \pi_{\mathsf{Inc},n})$
transaction id
$\mathsf{local\_state\_root} \leftarrow \mathsf{HASH}(\mathsf{id}_1, \ldots, \mathsf{id}_n)$
transitions
$\mathsf{t}_1$ $\mathsf{t}_2$ $\cdots$ $\mathsf{t}$ $\cdots$ $\mathsf{t}_{n-1}$ $\mathsf{t}_n$
transition
transition id
program id
function name
inputs
outputs
transition public key
transition commitment
proof
t
id
pid
fn
inputs
outputs
tpk
tcm
$\pi_t$

**Figure 3:** Diagram of an execution transaction (unbatched).

### 5.5.1 Executions

An *execution*, denoted by $\mathsf{ex}$, is a transaction type which consists of multiple transitions, a public instance $\mathbb{x}_{\mathsf{ex}}$ for the execute relation $\mathcal{R}_{\mathsf{ex}}$ (see Section 6.1), and a zero-knowledge proof for $\mathcal{R}_{\mathsf{ex}}$. The top-most transition in an execution is designated as the *root* transition, from which all other transitions (or function calls) stem. Optionally, the last transition is reserved for the transaction fee.

In an execution, a user must prove the following properties, discussed formally in Section 6.1:

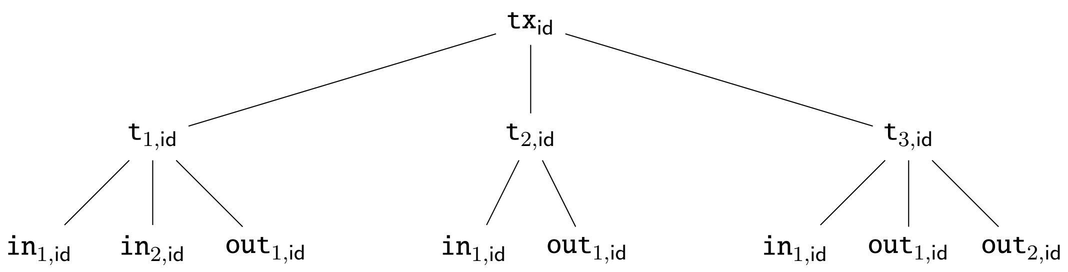
- *Every non-ephemeral input record serial number corresponds to a commitment that is contained in the ledger. Furthermore, this serial number is unique and must not already be present on the ledger (to prevent double-spends).*
- *Every output record commitment corresponds to a unique serial number.*
- *The input records and output records are consumed or created properly, corresponding to the rules specified in $f$.*

- *If the proof computation is delegated to an untrusted worker, the requests for the function execution authenticate correctly under the user's compute key (see Section 5.6).*

Note that due to the cryptographic hiding property of $\mathsf{CM}$ and the pseudorandomness of the $\mathsf{PRF}$, the record commitment and its serial number cannot be linked and *reveal no information* about the record attributes (Section 5.2). Formally, an *execution transaction* is a tuple $\mathsf{ex} = (\boldsymbol{T}, \mathbb{x}_{\mathsf{ex}}, \pi)$ where:

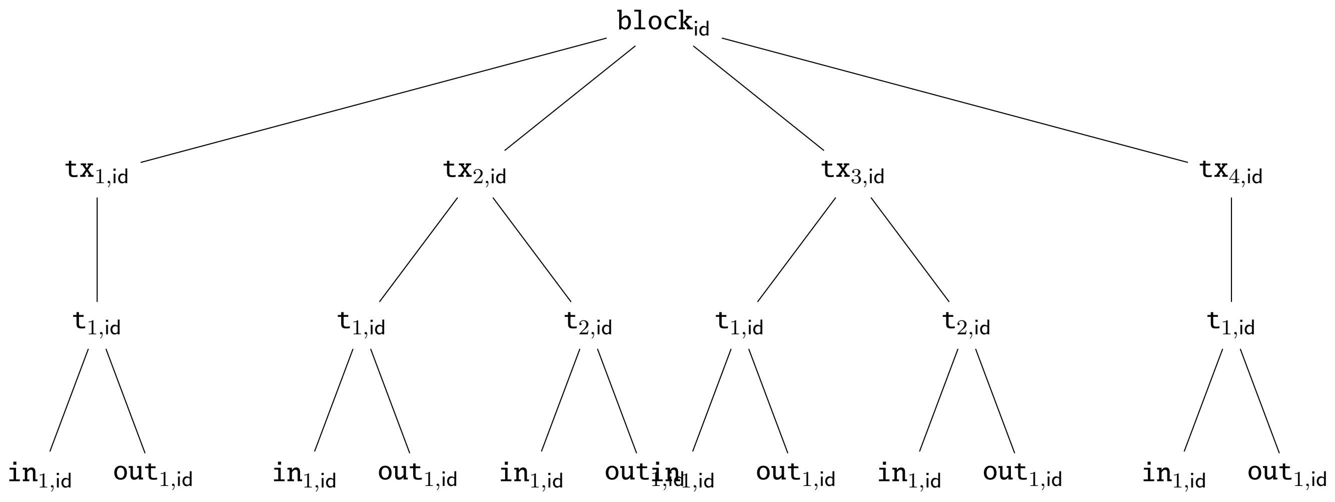
- $\boldsymbol{T}$ is a list of the transitions being executed, where the top-most function is referred to as the *root function* and the ordering is the same as that of a depth-first traversal of a call graph.
- $\mathbb{x}_{\mathsf{ex}}$ is an instance for the execute relation $\mathcal{R}_{\mathsf{ex}}$ (Definition 6.5). Note that the global state root $\mathcal{G}_{\mathsf{state}}$ is included in $\mathbb{x}_{\mathsf{ex}}$.
- $\pi$ is a proof that for the execute relation $\mathcal{R}_{\mathsf{ex}}$.

We consider preprocessing arguments that are universal and publically verifiable, so the proof $\pi$ can be publically verified and its circuit keys can be deterministically generated by any node in the network.



$\mathtt{tx}_{\mathsf{id}}$
$\mathtt{t}_{1,\mathsf{id}}$
$\mathtt{t}_{2,\mathsf{id}}$
$\mathtt{t}_{3,\mathsf{id}}$
$\mathtt{in}_{1,\mathsf{id}}$
$\mathtt{in}_{2,\mathsf{id}}$
$\mathtt{out}_{1,\mathsf{id}}$
$\mathtt{in}_{1,\mathsf{id}}$
$\mathtt{out}_{1,\mathsf{id}}$
$\mathtt{in}_{1,\mathsf{id}}$
$\mathtt{out}_{1,\mathsf{id}}$
$\mathtt{out}_{2,\mathsf{id}}$

**Figure 4:** The Merkilization of a transaction with three transitions of variable input/output records. $\mathtt{tx}_{\mathsf{id}}$ is also known as the *local state root*.



$\mathtt{block}_{\mathsf{id}}$
$\mathtt{tx}_{1,\mathsf{id}}$
$\mathtt{tx}_{2,\mathsf{id}}$
$\mathtt{tx}_{3,\mathsf{id}}$
$\mathtt{tx}_{4,\mathsf{id}}$
$\mathtt{t}_{1,\mathsf{id}}$
$\mathtt{t}_{1,\mathsf{id}}$
$\mathtt{t}_{2,\mathsf{id}}$
$\mathtt{t}_{1,\mathsf{id}}$
$\mathtt{t}_{2,\mathsf{id}}$
$\mathtt{t}_{1,\mathsf{id}}$
$\mathtt{in}_{1,\mathsf{id}}$
$\mathtt{out}_{1,\mathsf{id}}$
$\mathtt{in}_{1,\mathsf{id}}$
$\mathtt{out}_{1,\mathsf{id}}$
$\mathtt{in}_{1,\mathsf{id}}$
$\mathtt{out}_{1,\mathsf{id}}$
$\mathtt{in}_{1,\mathsf{id}}$
$\mathtt{out}_{1,\mathsf{id}}$
$\mathtt{in}_{1,\mathsf{id}}$
$\mathtt{out}_{1,\mathsf{id}}$
$\mathtt{in}_{1,\mathsf{id}}$
$\mathtt{out}_{1,\mathsf{id}}$

**Figure 5:** The Merkilization of several transactions.

### 5.5.2 Deployments

A *deployment*, denoted by $\mathsf{dp}$, consists of program along with a mapping that associates each program function to its proving and verifying keys. In addition, a cryptographic certificate is provided for each function to authenticate that the proving and verification keys correspond to the function circuit which they claim to represent.

Formally, a deployment transaction is a tuple $\mathsf{dp} = (\mathcal{P}, \boldsymbol{M})$ where:

- $\mathcal{P}$ is the ambient program that contains the functions for which the circuit keys correspond to.
- $\boldsymbol{M} = [(\mathsf{fid}_i, (\mathsf{ipk}_i, \mathsf{ivk}_i, c_i))]_{i=1}^{n}$ is a mapping that associates a function id fid (Section 5.3.2) in the program $\mathcal{P}$ to a proving key ipk, verifying key ivk, and a certificate $c$.

In order to retrieve the deployment for $\mathcal{P}$, a user will invoke the ledger abstraction (Section 5.7) as $\mathsf{dp} \leftarrow \mathbf{L}.\mathsf{FetchDeployment}(\mathcal{P})$. For a function $f$ or its NP relation $\mathcal{R}_f$, we abuse notation to fetch the circuit keys as $(\mathsf{ipk}_f, \mathsf{ivk}_f) := \mathbf{L}.\mathsf{FetchDeployment}(\mathcal{R}_f)$.

## 5.6 Requests

A *request* enables a user to delegate to an untrusted worker, such as a remote server, the computation of a proof for the execute relation $\mathcal{R}_{\mathsf{ex}}$ (Definition 6.5). The security guarantee, informally, *is that the worker should be unable to produce transactions on behalf of the user that the user did not authorize.* At a high level, inside a request, the user communicates the instance-witness tuple $(\mathbb{x}_{\mathsf{ex}}, \mathbb{w}_{\mathsf{ex}})$ for $\mathcal{R}_{\mathsf{ex}}$ to the worker who produces the proof by invoking $\mathsf{NIZK.Prove}(\mathsf{ipk}, \mathbb{x}_{\mathsf{ex}}, \mathbb{w}_{\mathsf{ex}})$.

While an individual request corresponds to the execution of a singular transition, requests are sent in batches known as *authorizations*. More precisely, an authorization is an ordered list of requests, the top-most request representing a “request” to execute the root transition, with the subsequent transitions being ordered in a depth-first manner over the call graph.

Note that the worker is able to prove properties about records, such as commitment openings and serial number derivations, without the account private key ask. Crucially, the user only sends to the worker the account compute key ack along with other witness elements, such as the commitment randomness, to enable to worker to produce proofs for $\mathcal{R}_{\mathsf{ex}}$ over selected choices of records.

Formally, a request is a tuple:

$$\mathsf{req} = (\mathsf{apk}, \mathcal{P}, f, [\mathsf{id}_i]_1^N, \mathsf{tvk}, \mathsf{scm}, \mathsf{tcm}, \sigma)$$

where:

- apk is the account address of the user issuing the request.
- $\mathcal{P}$ is a program that contains the function that the request executes.
- $f$ is a function for the NP relation $\mathcal{R}_f$ which the request executes.
- $[\mathsf{id}_i]_1^N$ is a list of input ids. The total number of inputs $N = n + s_1 + s_2$ means there are $n$ input records, $s_1$ non-record public inputs, and $s_2$ non-record private inputs. For $i \in \{1, \ldots, n\}$, $\mathsf{id}_i$ is parsed as $(\mathsf{i}_i, \mathsf{sn}_i, \mathsf{cm}_i)$, whereas for $k \in \{n+1, \ldots, s_1 + s_2\}$, $\mathsf{id}_k$ is parsed as $(\mathsf{i}_k, c_k)$.
- tvk is the transition view key used to encrypt private transition inputs and outputs and generate the transition commitment. See Section 5.4 for more details.
- tcm is the transition commitment and is generated by committing to the tvk. See Section 5.4 for more details.
- scm is the consistency signer commitment, used to enforce that all of the child requests in the ambient authorization are executed by the worker. In more detail, in the root request of an authorization, the scm is computed as $\mathsf{scm} := \mathsf{CM.Commit}(\mathsf{pp}_{\mathsf{CM}}, \mathsf{apk}||\mathsf{tcm}_1|| \ldots ||\mathsf{tcm}_d)$, where $\mathsf{tcm}_i$ corresponds to the transition commitment of a child request in the order of execution.
- $\sigma$ is a signature of the request contents parsed as $m = (\mathsf{addr}, \mathsf{fid}, [\mathsf{id}_i]_1^N, \mathsf{tvk}, \mathsf{tcm}, \mathsf{scm})$.

## 5.7 Ledger

For the sake of explanation, the ledger $\mathbf{L}$ in our prototype is simply an ideal ledger, i.e., an append-only log of valid transactions that is stored in memory. Of course, in a real-world deployment, this ideal ledger would be replaced by a distributed protocol that realizes (a suitable approximation of) an ideal ledger. Recall that we require require the ledger $\mathbf{L}$ to provide a method to efficiently prove and verify membership of a transaction, or one of its subcomponents, in $\mathbf{L}$. For this, we maintain a Merkle tree atop the list of transactions, using the collision-resistant hash function $\mathsf{CRH}$ (Section 3). This results in the following algorithms for $\mathbf{L}$.

- $\mathbf{L}.\mathsf{Push}(\mathsf{tx})$: Append $\mathsf{tx}$ to the transaction list and update the Merkle tree.
- $\mathbf{L}.\mathsf{Digest} \to \mathsf{st}_{\mathbf{L}}$: Return the root of the Merkle tree, also known as the global state root.
- $\mathbf{L}.\mathsf{Prove}(\mathsf{tx}) \to \mathbb{w}_{\mathbf{L}}$: Return the authentication path for $\mathsf{tx}$ in the Merkle tree.
- $\mathbf{L}.\mathsf{Verify}(\mathsf{st}_{\mathbf{L}}, \mathsf{tx}, \mathbb{w}_{\mathbf{L}}) \to b$: Check that $\mathbb{w}_{\mathbf{L}}$ is a valid authentication path for $\mathsf{tx}$ in a tree with root $\mathsf{st}_{\mathbf{L}}$.

Our prototype maintains the Merkle tree in memory, but a real-world deployment would have to maintain it via a distributed protocol. Furthermore, to the ledger $\mathbf{L}$, we add the following functionality to reflect that $\mathsf{NP}$ statements are proven using universal $\mathsf{NIZK}$ arguments.

- $\mathbf{L}.\mathsf{UniversalParameters} \to \mathsf{pp}_{\mathsf{U}}$: Return the universal public parameters $\mathsf{pp}_{\mathsf{U}}$ for proving computations.
- $\mathbf{L}.\mathsf{UpdateMappings}(\mathsf{addr}, v) \to b$: Update the globally persistent mapping to hold the value $v$ when indexed at the key $\mathsf{addr}$.
- $\mathbf{L}.\mathsf{FetchDeployment}(\mathcal{P}) \to \mathsf{dp}$: Return the deployment, that is, the circuit-specific keys for the constituent program functions, for a specified input program $\mathcal{P}$.

# 6 Algorithms for constructions of Aleo DPC schemes

## 6.1 $\mathsf{NP}$ relations for state execution

In this section, we describe $\mathsf{NP}$ relations for the following tasks:

- **Request verification.** Aside from containing a proof attesting that $f$, when evaluated on the set of prescribed inputs, produces the set of claimed outputs, a transition contains a proof of correct request verification. In more detail, when a user communicates the necessary secrets to a worker, in the form of a request, to delegate a proof computation, the worker must prove that indeed the request authenticates correctly under the account compute key of the user issuing the request. Due to the unforgeability property of signature schemes, a passing signature verification ensures this condition holds. In addition, the $\mathsf{NP}$ relation should also ensure that the request object itself is well-formed, i.e., that the inputs to the request match their claimed commitments. This relation, denoted by $\mathcal{R}_{\mathsf{req}}$, is formalized in Definition 6.1.

- **Authorization verification.** A single authorization may contain several requests corresponding to function call invocations from the root function. For secure delegation, we need to ensure that an adversarial worker is prevented from peeling off layers of execution by only providing proofs for a subset of the requests. To do so, we provide an additional relation $\mathcal{R}_{\mathsf{auth}}$ that enforces the well-formedness of each individual request, but moreover enforces that the the signer commitment of the root request is consistent with that of its children requests. The relation $\mathcal{R}_{\mathsf{auth}}$ is formalized in Definition 6.2 and utilizes $\mathcal{R}_{\mathsf{req}}$ as a subroutine.

- **Local inclusion verification.** Each execution transaction contains a local state root, which is a commitment to the inputs and outputs of the constituent transitions. When an execution is pushed to the ledger, the global ledger digest is updated using the local state root of the execution. Thus, verifying an execution must entail verifying the Merkle tree inclusions of these commitments in the tree represented by the local state root. This relation, denoted by $\mathcal{R}_{\mathsf{loc}}$, is formalized in Definition 6.3.

- **Global inclusion verification.** A valid execution transaction must properly consume records that are maintained on the ledger, which is demonstrated by providing a ledger membership witness for the commitment to each input record. Furthermore, a valid execution only contains input records which have not been previously consumed. This can be achieved by publishing a unique serial number for every input record being consumed. A deterministic check on the record nonce computation for *output records* enforces the foregoing uniqueness property. This relation, denoted by $\mathcal{R}_{\mathsf{glb}}$, is formalized in Definition 6.4.

The foregoing $\mathsf{NP}$ relations are used to construct the grand execution relation $\mathcal{R}_{\mathsf{ex}}$ in Definition 6.5.

**Definition 6.1** (Request verification relation)**.** *The* $\mathsf{NP}$ *relation* $\mathcal{R}_{\mathsf{req}}$ *is the set of all tuples* $(\mathbb{x}, \mathbb{w})$

$$\mathbb{x} = \begin{pmatrix} \textit{program and function} & (\mathcal{P}, f) \\ \textit{serial numbers of input records} & [\mathsf{sn}_i]_1^n \\ \textit{input IDs of non-records} & [\mathsf{id}_j]_1^s \\ \textit{public non-record inputs} & [\mathsf{i}_k^{\mathsf{pub}}]_1^{s_1} \\ \textit{transition commitment} & \mathsf{tcm} \\ \textit{transition public key} & \mathsf{tpk} \\ \textit{signature and message} & (\sigma, m) \end{pmatrix}, \mathbb{w} = \begin{pmatrix} \textit{account compute key} & \mathsf{ack} \\ \textit{commitments of input records} & [\mathsf{cm}_i]_1^n \\ \textit{input records} & [\mathbf{r}_i]_1^n \\ \textit{private non-record inputs} & [\mathsf{i}_k^{\mathsf{priv}}]_1^{s_2} \\ \textit{input view keys} & [\mathsf{ivk}_k]_1^s \\ \textit{transition view key} & \mathsf{tvk} \end{pmatrix}$$

*that satisfy the following conditions.*

```
$\mathcal{R}_{\mathsf{req}}(\mathbb{x}, \mathbb{w})$
1. Parse the compute key $\mathsf{ack}$ as $(\mathsf{pk}_{\mathsf{SIG}}, \mathsf{pr}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{PRF}})$.
2. Derive the account public key: $\mathsf{apk} := \mathcal{A}.\mathsf{GenAddress}(\mathsf{ack})$.
3. Check that the transition commitment is valid: $\mathsf{tcm} = \mathsf{CM}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{CM}}, \mathsf{tvk})$.
4. Check the message is valid: $m = (\mathsf{pk}_{\mathsf{SIG}}, \mathsf{pr}_{\mathsf{SIG}}, \mathsf{apk}, \mathsf{tcm}, \mathsf{fid}, [\mathsf{sn}_i]_1^n, [\mathsf{id}_j]_1^s)$.
5. Check that the signature is valid: $\sigma = \mathsf{SIG}.\mathsf{Verify}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{SIG}}, m, \sigma)$.
6. For each $i \in \{1, \dots, n\}$:
   • Parse the record $\mathbf{r}_i$ as $(v_i, \mathsf{apk}_i, \mathsf{d}_i, \rho_i)$.
   • Check the owner address $\mathsf{apk}_i$ is valid: $\mathsf{apk}_i = \mathsf{apk}$.
   • Check the serial number $\mathsf{sn}_i$ is valid: $\mathsf{sn}_i := \mathsf{PRF}^{\mathsf{SN}}(\mathsf{sk}_{\mathsf{PRF}}, \rho_i)$.
   • Check the record commitment $\mathsf{cm}_i$ is valid: $\mathsf{cm}_i := \mathsf{CM}^{\mathsf{R}}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{CM}}, v_i || \mathsf{apk}_i || \mathsf{d}_i || \rho_i; r_i)$.
7. For each $j \in \{1, \dots, s\}$:
   • Parse the non-record input $\mathsf{i}_j$ as $(v_j, \mathsf{d}_j)$.
   • If $v_j$ is public, check the input id $\mathsf{id}_j$ is valid: $\mathsf{id}_j := \mathsf{CRH}.\mathsf{Eval}(\mathsf{fid}, \mathsf{tvk}, j, \mathsf{d}_j)$.
   • If $v_j$ is private, check the input id $\mathsf{id}_j$ is valid: $\mathsf{d}_j = \mathsf{ENC}.\mathsf{Dec}(\mathsf{pp}_{\mathsf{ENC}}, \mathsf{ivk}_j, c_j)$.
```

**Definition 6.2** (Authorization relation)**.** *The* NP *relation* $\mathcal{R}_{\mathsf{auth}}$ *is the set of all tuples* $(\mathbb{x}, \mathbb{w})$ *parsed as* $\mathbb{x} = (\mathbb{x}_{\mathsf{root}}, [\mathbb{x}_{\mathsf{req},i}]_1^d, \mathbb{x}_{f_{\mathsf{root}}}, [\mathbb{x}_{f_i}]_1^d)$ *and* $\mathbb{w} = (\mathbb{w}_{\mathsf{root}}, [\mathbb{w}_{\mathsf{req},i}]_1^d, \mathbb{w}_{f_{\mathsf{root}}}, [\mathbb{w}_{f_i}]_1^d)$ *that satisfy the following conditions. Recall Definition 5.1 for a description of* $\mathcal{R}_f$ *given a function* $f$*.*

```
$\mathcal{R}_{\mathsf{auth}}(\mathbb{x}, \mathbb{w})$:
1. The individual requests are well-formed: $\mathcal{R}_{\mathsf{req}}(\mathbb{x}_{\mathsf{root}}, \mathbb{w}_{\mathsf{root}}) \wedge \bigwedge_{i=1}^{d} \mathcal{R}_{\mathsf{req}}(\mathbb{x}_{\mathsf{req},i}, \mathbb{w}_{\mathsf{req},i}) = 1$.
2. Check the function predicates are satisfied: $\mathcal{R}_{f_{\mathsf{root}}}(\mathbb{x}_{f_{\mathsf{root}}}, \mathbb{w}_{f_{\mathsf{root}}}) \wedge \bigwedge_{i=1}^{d} \mathcal{R}_{f_i}(\mathbb{x}_{f_i}, \mathbb{w}_{f_i}) = 1$.
3. Retrieve the transition signer commitment $\mathsf{scm}_{\mathsf{root}}$ from $\mathbb{x}_{\mathsf{root}}$.
4. For each $i \in \{1, \dots, d\}$, retrieve the transition commitment $\mathsf{tcm}_{\mathsf{req},i}$ from $\mathbb{x}_{\mathsf{req},i}$.
5. Check the child requests are linked to the root: $\mathsf{scm}_{\mathsf{root}} = \mathsf{CRH}.\mathsf{Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{tcm}_{\mathsf{req},1} || \dots || \mathsf{tcm}_{\mathsf{req},1})$.
6. Check that the instances of $\mathcal{R}_{\mathsf{req}}$ and $\mathcal{R}_f$ are consistent. For each $i \in \{1, \dots, d\}$:
   • Parse $\mathbb{x}_{f_i}$ as $(\mathcal{P}, f, [\mathsf{sn}_i]_1^n, [\mathsf{cm}'_i]_1^m, [\mathsf{id}_j]_1^s, [\mathsf{id}'_j]_1^t, [\mathsf{i}_k^{\mathsf{pub}}]_1^{s_1})$.
   • Parse $\mathbb{x}_{\mathsf{req},i}$ as $(\mathcal{P}*, f*, [\mathsf{sn}_i]_1^n*, [\mathsf{id}_j]_1^s*, [\mathsf{i}_k^{\mathsf{pub}}]_1^{s_1}*, \mathsf{tcm}*, \mathsf{tpk}*, \sigma*, m*)$.
   • Check the input serial numbers are consistent: $[\mathsf{sn}_i]_1^n = [\mathsf{sn}_i]_1^n*$.
   • Check the input IDs of the non-record inputs are consistent: $[\mathsf{id}_j]_1^s = [\mathsf{id}_j]_1^s*$.
```

**Definition 6.3** (Local inclusion relation)**.** *The* NP *relation* $\mathcal{R}_{\mathsf{loc}}$ *is the set of all tuples* $(\mathbb{x}, \mathbb{w})$

$$\mathbb{x} = \left( \begin{array}{ll} \textit{local state root} & \ell_{\mathsf{state}} \\ \textit{serial numbers of input records} & [\mathsf{sn}_i]_1^n \\ \textit{commitments of output records} & [\mathsf{cm}'_i]_1^m \end{array} \right), \mathbb{w} = \left( \begin{array}{ll} \textit{account compute key} & \mathsf{ack} \\ \textit{input records} & [\mathbf{r}_i]_1^n \\ \textit{commitments of input records} & [\mathsf{cm}_i]_1^n \\ \textit{input record membership witnesses} & [\mathbb{w}_{\mathbf{L},i}]_1^n \\ \textit{output records} & [\mathbf{r}'_i]_1^m \\ \textit{output record membership witnesses} & [\mathbb{w}'_{\mathbf{L},i}]_1^m \end{array} \right)$$

*that satisfy the following conditions.*

```
$\mathcal{R}_{\mathsf{loc}}(\mathbb{x}, \mathbb{w})$:
1. Parse the compute key $\mathsf{ack}$ as $(\mathsf{pk}_{\mathsf{SIG}}, \mathsf{pr}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{PRF}})$.
2. Derive the account public key: $\mathsf{apk} := \mathcal{A}.\mathsf{GenAddress}(\mathsf{ack})$.
3. Check that the input records are contained in the transaction. For each $i \in \{1, \ldots, n\}$:
    • Parse $\mathbf{r}_i$ as $(v_i, \mathsf{apk}_i, \mathsf{d}_i, \rho_i)$.
    • Check the owner address $\mathsf{apk}_i$ is valid: $\mathsf{apk}_i = \mathsf{apk}$.
    • Check the serial number $\mathsf{sn}_i$ is valid: $\mathsf{sn}_i := \mathsf{PRF}^{\mathsf{SN}}(\mathsf{sk}_{\mathsf{PRF}}, \rho_i)$.
    • Check the record commitment $\mathsf{cm}_i$ is valid: $\mathsf{cm}_i := \mathsf{CM}^{\mathsf{R}}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{CM}}, v_i || \mathsf{apk}_i || \mathsf{d}_i || \rho_i; r_i)$.
    • Check the membership of $\mathsf{cm}_i$ in the local state with digest $\ell_{\mathsf{state}}$: $\mathsf{T}.\mathsf{Verify}(\ell_{\mathsf{state}}, \mathsf{cm}_i, \mathbb{w}_{\mathbf{L},i}) = 1$.
4. Check that the output records are contained in the transaction. For each $j \in \{1, \ldots, m\}$:
    • Parse $\mathbf{r}'_j$ as $(v'_j, \mathsf{apk}'_j, \mathsf{d}'_j, \rho'_j, r'_j)$.
    • Check the owner address $\mathsf{apk}'_j$ is valid: $\mathsf{apk}'_j = \mathsf{apk}$.
    • Check the record commitment $\mathsf{cm}'_j$ is valid: $\mathsf{cm}'_j := \mathsf{CM}^{\mathsf{R}}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{CM}}, v'_j || \mathsf{apk}'_j || \mathsf{d}'_j || \rho'_j; r'_j)$.
    • Check the membership of $\mathsf{cm}'_j$ in the local state with digest $\ell_{\mathsf{state}}$: $\mathsf{T}.\mathsf{Verify}(\ell_{\mathsf{state}}, \mathsf{cm}'_j, \mathbb{w}'_{\mathbf{L},j}) = 1$.
```

**Definition 6.4** (Global inclusion relation)**.** *The* NP *relation $\mathcal{R}_{\mathsf{glb}}$ is the set of all tuples $(\mathbb{x}, \mathbb{w})$*

$$\mathbb{x} = \begin{pmatrix} \textit{global state root} & \mathcal{G}_{\mathsf{state}} \\ \textit{serial numbers of input records} & [\mathsf{sn}_i]_1^n \\ \textit{commitments of output records} & [\mathsf{cm}'_i]_1^m \end{pmatrix}, \mathbb{w} = \begin{pmatrix} \textit{account compute key} & \mathsf{ack} \\ \textit{input records} & [\mathbf{r}_i]_1^n \\ \textit{commitments of input records} & [\mathsf{cm}_i]_1^n \\ \textit{input record membership witnesses} & [\mathbb{w}_{\mathbf{L},i}]_1^n \\ \textit{output records} & [\mathbf{r}'_i]_1^m \\ \textit{output record membership witnesses} & [\mathbb{w}'_{\mathbf{L},i}]_1^m \end{pmatrix}$$

*that satisfy the following conditions.*

```
$\mathcal{R}_{\mathsf{glb}}(\mathbb{x}, \mathbb{w})$:
1. Parse the compute key $\mathsf{ack}$ as $(\mathsf{pk}_{\mathsf{SIG}}, \mathsf{pr}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{PRF}})$.
2. Derive the account public key: $\mathsf{apk} := \mathcal{A}.\mathsf{GenAddress}(\mathsf{ack})$.
3. Check that the input records are contained in the transaction. For each $i \in \{1, \ldots, n\}$:
    • Parse $\mathbf{r}_i$ as $(v_i, \mathsf{apk}_i, \mathsf{d}_i, \rho_i)$.
    • Check the owner address $\mathsf{apk}_i$ is valid: $\mathsf{apk}_i = \mathsf{apk}$.
    • Check the serial number $\mathsf{sn}_i$ is valid: $\mathsf{sn}_i := \mathsf{PRF}^{\mathsf{SN}}(\mathsf{sk}_{\mathsf{PRF}}, \rho_i)$.
    • Check the record commitment $\mathsf{cm}_i$ is valid: $\mathsf{cm}_i := \mathsf{CM}^{\mathsf{R}}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{CM}}, v_i || \mathsf{apk}_i || \mathsf{d}_i || \rho_i; r_i)$.
    • Check the membership of $\mathsf{cm}_i$ in the local state with digest $\mathcal{G}_{\mathsf{state}}$: $\mathbf{L}.\mathsf{Verify}(\ell_{\mathsf{state}}, \mathsf{cm}_i, \mathbb{w}_{\mathbf{L},i}) = 1$.
4. Check that the output records are contained in the transaction. For each $j \in \{1, \ldots, m\}$:
    • Parse $\mathbf{r}'_j$ as $(v'_j, \mathsf{apk}'_j, \mathsf{d}'_j, \rho'_j, r'_j)$.
    • Check the owner address $\mathsf{apk}'_j$ is valid: $\mathsf{apk}'_j = \mathsf{apk}$.
    • Check the record commitment $\mathsf{cm}'_j$ is valid: $\mathsf{cm}'_j := \mathsf{CM}^{\mathsf{R}}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{CM}}, v'_j || \mathsf{apk}'_j || \mathsf{d}'_j || \rho'_j; r'_j)$.
    • Check the membership of $\mathsf{cm}'_j$ in the local state with digest $\mathcal{G}_{\mathsf{state}}$: $\mathbf{L}.\mathsf{Verify}(\ell_{\mathsf{state}}, \mathsf{cm}'_j, \mathbb{w}'_{\mathbf{L},j}) = 1$.
```

**Definition 6.5** (Execute relation)**.** *The* NP *relation $\mathcal{R}_{\mathsf{ex}}$ is the set of all tuples $(\mathbb{x}, \mathbb{w})$ parsed as $\mathbb{x} = (\mathbb{x}_{\mathsf{auth}}, \mathbb{x}_{\mathsf{loc}}, \mathbb{x}_{\mathsf{glb}})$ and $\mathbb{w} = (\mathbb{w}_{\mathsf{auth}}, \mathbb{x}_{\mathsf{loc}}, \mathbb{w}_{\mathsf{glb}})$ that satisfy the following conditions.*

$\mathcal{R}_{\mathsf{ex}}(\mathbb{x}, \mathbb{w})$ :

1. Check that the authorization verification relation $\mathcal{R}_{\mathsf{auth}}$ is satisfied: $\mathcal{R}_{\mathsf{auth}}(\mathbb{x}_{\mathsf{auth}}, \mathbb{w}_{\mathsf{auth}}) = 1$.
2. Check that the local state inclusion relation $\mathcal{R}_{\mathsf{loc}}$ is satisfied: $\mathcal{R}_{\mathsf{loc}}(\mathbb{x}_{\mathsf{loc}}, \mathbb{x}_{\mathsf{loc}}) = 1$.
3. Check that the global state inclusion relation $\mathcal{R}_{\mathsf{glb}}$ is satisfied: $\mathcal{R}_{\mathsf{glb}}(\mathbb{x}_{\mathsf{glb}}, \mathbb{w}_{\mathsf{glb}}) = 1$.
4. Parse $\mathbb{x}_{\mathsf{loc}}$ as $(\mathbb{x}_{\mathsf{loc},1}, \ldots, \mathbb{x}_{\mathsf{loc},d})$.
5. Parse $\mathbb{x}_{\mathsf{glb}}$ as $(\mathbb{x}_{\mathsf{glb},1}, \ldots, \mathbb{x}_{\mathsf{glb},d})$.
6. Parse $\mathbb{x}_{\mathsf{auth}}$ as $(\mathbb{x}_{\mathsf{req},1}, \ldots, \mathbb{x}_{\mathsf{req},d}, \mathbb{x}_{f_1}, \ldots, \mathbb{x}_{f_d})$.
7. Check that the instances of $\mathcal{R}_{\mathsf{loc}}$ and $\mathcal{R}_{\mathsf{glb}}$ are consistent as follows. For each $i \in \{1, \ldots, d\}$:
   - Parse $\mathbb{x}_{\mathsf{glb},i}$ as $(\mathcal{G}_{\mathsf{state}}, [\mathsf{sn}_i]_1^n, [\mathsf{cm}'_i]_1^m)$.
   - Parse $\mathbb{x}_{\mathsf{loc},i}$ as $(\ell_{\mathsf{state}}, [\mathsf{sn}_i]_1^{n,*}, [\mathsf{cm}'_i]_1^{m,*})$.
   - Parse $\mathbb{x}_{f_i}$ as $(\mathcal{P}, f, [\mathsf{sn}_i]_1^{n,\dagger}, [\mathsf{cm}'_i]_1^{m,\dagger}, [\mathsf{id}_j]_1^s, [\mathsf{id}'_j]_1^t, [\mathsf{i}_k^{\mathsf{pub}}]_1^{s_1})$.
   - Check that the input serial numbers are consistent: $[\mathsf{sn}_i]_1^n = [\mathsf{sn}_i]_1^{n,*} = [\mathsf{sn}_i]_1^{n,\dagger}$.
   - Check that the output commitments are consistent: $[\mathsf{cm}'_i]_1^m = [\mathsf{cm}'_i]_1^{m,*} = [\mathsf{cm}'_i]_1^{m,\dagger}$.

## 6.2 Virtual machines

We define the notion of a *virtual machine* (VM), which refers to an abstract computing environment that operates in a controlled and isolated manner. In real-world use-cases, many DPC schemes are implemented using VM constructions that abstract away computing resources like CPU, memory-management, and storage. Furthermore, in Section 6.3, we provide a concrete construction of a VM in the ALEO DPC model outlined in Section 4. In our abstraction, we refer to $M$ as the *machine state* that algorithms associated to the VM can internally update.

Formally, we define the virtual machine as a tuple of algorithms:

$$\mathsf{VM} = (\mathsf{Setup}, \mathsf{Authorize}, \mathsf{Execute}, \mathsf{Finalize}, \mathsf{Synthesize}, \mathsf{VfyExec}, \mathsf{VfyDeploy})$$

with the following syntax.

- $\mathsf{VM.Setup}^{\mathbf{L}}(1^\lambda) \to (\mathsf{pp}, \mathsf{ask}, \mathsf{ack}, \mathsf{avk}, \mathsf{apk})$: On input query access to a ledger $\mathbf{L}$ and a security parameter $\lambda$ (in unary), $\mathsf{VM.Setup}$ samples the public parameters $\mathsf{pp}$, an account private key $\mathsf{ask}$, compute key $\mathsf{ack}$, view key $\mathsf{avk}$, and public key $\mathsf{apk}$.

- $\mathsf{VM.Authorize}^{\mathbf{L}}(\mathsf{pp}, \mathsf{ask}, \mathcal{P}, f, [\mathsf{i}_j]_1^n, M) \to (\mathsf{auth}, M')$ : On input query access to a ledger $\mathbf{L}$ and machine state $M$, an account private key $\mathsf{ask}$, a program $\mathcal{P}$, a program function $f$, and function inputs $[\mathsf{i}_j]_1^n$, $\mathsf{VM.Authorize}$ outputs a list of authorized requests $\mathsf{auth}$ and the updated machine state $M'$. If $f$ does not belong to $\mathcal{P}$ or if for any $k \in [n]$, $\mathsf{i}_k$ is not a valid input to $f$, output $\perp$.

- $\mathsf{VM.Execute}^{\mathbf{L}}(\mathsf{pp}, \mathsf{ack}, \mathsf{auth}, M) \to (\mathsf{ex}, M')$: On input query access to a ledger $\mathbf{L}$, machine state $M$, an account compute key $\mathsf{ack}$, and a list of authorized requests $\mathsf{auth}$, $\mathsf{VM.Execute}$ outputs the updated machine state $M'$ and an execution $\mathsf{ex}$.

- $\mathsf{VM.Finalize}^{\mathbf{L}}(\mathsf{pp}, \boldsymbol{T}, M) \to M'$: On input query access to a ledger $\mathbf{L}$, machine state $M$, and a list of transitions $\boldsymbol{T}$, $\mathsf{VM.Finalize}$ outputs the update machine state $M'$ by modifying persistent mappings maintained on the ledger $\mathbf{L}$.

- $\mathsf{VM.Synthesize}^{\mathbf{L}}(\mathsf{pp}, \mathcal{P}, M) \to (\mathsf{dp}, M')$: On input query access to a ledger $\mathbf{L}$, machine state $M$ and and program $\mathcal{P}$, $\mathsf{VM.Synthesize}$ outputs a program deployment $\mathsf{dp}$ and the updated machine state $M'$.

- $\mathsf{VM.VfyExec}^{\mathbf{L}}(\mathsf{pp}, \mathsf{avk}, \mathsf{ex}, M) \to (b, M')$: On input query access to a ledger $\mathbf{L}$, machine state $M$, an account view key $\mathsf{avk}$, and an execution $\mathsf{ex}$, $\mathsf{VM.VfyExec}$ outputs a decision bit $b$ indicating that the execution $\mathsf{ex}$ is well-formed relative to $\mathcal{R}_{\mathsf{ex}}$ and the altered machine state $M'$.
- $\mathsf{VM.VfyDeploy}^{\mathbf{L}}(\mathsf{pp}, \mathsf{dp}, M) \to (b, M')$: On input query access to a ledger $\mathbf{L}$, machine state $M$ and a deployment $\mathsf{dp}$, $\mathsf{VM.VfyDeploy}$ will output a decision bit $b$ indicating that $\mathsf{dp}$ is well-formed and the altered machine state $M'$.

## 6.3 VM constructions for Aleo DPC schemes

### 6.3.1 Setup

```
$\mathsf{VM.Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{ask}, \mathsf{ack}, \mathsf{avk}, \mathsf{apk})$ :
 1. Sample the universal public parameters for NIZK: $\mathsf{pp_U} := \mathbf{L}.\mathsf{UniversalParameters}$.
 2. Sample the signature public parameters: $\mathsf{pp_{SIG}} \leftarrow \mathsf{SIG.Setup}(1^\lambda)$.
 3. Sample the commitment public parameters: $\mathsf{pp_{CM}} \leftarrow \mathsf{CM.Setup}(1^\lambda)$.
 4. Sample the hash function public parameters: $\mathsf{pp_{CRH}} \leftarrow \mathsf{CRH.Setup}(1^\lambda)$.
 5. Sample the signature key pair $(\mathsf{sk_{SIG}}, \mathsf{pk_{SIG}}) \leftarrow \mathsf{SIG.Keygen}(\mathsf{pp_{SIG}})$.
 6. Randomly sample the signature randomizer $r_{\mathsf{SIG}} \leftarrow \mathbb{F}$.
 7. Set the account private key $\mathsf{ask} := (\mathsf{sk_{SIG}}, r_{\mathsf{SIG}})$.
 8. Generate the account compute key, view key, and address: $(\mathsf{ack}, \mathsf{avk}, \mathsf{apk}) \leftarrow \mathcal{A}.\mathsf{GenAccount}(\mathsf{ask})$.
 9. Set the public parameters $\mathsf{pp} := (\mathsf{pp_U}, \mathsf{pp_{SIG}}, \mathsf{pp_{CM}}, \mathsf{pp_{CRH}})$.
10. Output $(\mathsf{pp}, \mathsf{ask}, \mathsf{ack}, \mathsf{avk}, \mathsf{apk})$.
```

### 6.3.2 Authorizations

$\mathsf{VM.Authorize}$ produces a list of authorized requests, or simply an authorization, for a root program function. Note this function may invoke external program functions as its instructions are executed. As such, there is one request per function call in the authorization, in the order in which the calls are made. $\mathsf{VM.Authorize}$ invokes the following auxiliary method to create a new request for each function call, where $\mathsf{root}$ is a boolean indicating if $f$ is a root function call.

```
$\mathsf{VM.ConstructRequest}(M, \mathsf{ask}, \mathcal{P}, f, [\mathsf{i}_j]_1^n, \mathsf{root}) \to \mathsf{req}$:
 1. Derive the account keys $(\mathsf{ack}, \mathsf{avk}, \mathsf{apk}) := \mathcal{A}.\mathsf{GenAccount}(\mathsf{ask})$.
 2. Parse the account private key ask as $(\mathsf{sk_{SIG}}, r_{\mathsf{SIG}})$.
 3. Parse the account compute key ack as $(\mathsf{pk_{SIG}}, \mathsf{pr_{SIG}}, \mathsf{sk_{PRF}})$.
 4. Sample a random nonce $\rho \leftarrow \mathbb{F}$.
 5. Compute the transition signing key as $\mathsf{tsk} := \mathsf{CRH.Eval}(\mathsf{pp_{CRH}}, \mathsf{sk_{SIG}}, \rho)$.
 6. Compute the transition public key $\mathsf{tpk} := \mathsf{CM.Commit}(\mathsf{pp_{CM}}, \mathsf{tsk})$.
 7. Compute the transition view key as $\mathsf{tvk} := \mathsf{CM.Commit}(\mathsf{pp_{CM}}, \mathsf{avk}; \mathsf{tsk})$.
 8. Compute the transition commitment as $\mathsf{tcm} := \mathsf{CM.Commit}(\mathsf{pp_{CM}}, \mathsf{tvk})$.
 9. Set the message encapsulating the request as $m := (\mathsf{pk_{SIG}}, \mathsf{pr_{SIG}}, \mathsf{apk}, \mathsf{rvk}, \mathsf{tvk}, \mathsf{tcm}, \mathsf{fid}, [\mathsf{i}_j]_1^n)$.
10. Sign the message $\sigma := \mathsf{SIG.Sign}(\mathsf{pp_{SIG}}, \mathsf{sk_{SIG}}, m; \mathsf{tsk})$.
11. For each input *of type record* $\mathsf{i}_j \in [\mathsf{i}_j]_1^n$, do as follows.
```

```
    (a) Parse $\mathbf{r}_j = \mathsf{i}_j$ as $(v_j, \mathsf{apk}_j, \mathsf{d}_j, \rho_j, r_j)$.
    (b) Compute the serial number $\mathsf{sn}_j := \mathsf{PRF}(\mathsf{sk_{PRF}}, \rho_j)$.
    (c) Compute the commitment $\mathsf{cm}_j := \mathsf{CM.Commit}(\mathsf{pp_{CM}}, v_j||\mathsf{apk}||\mathsf{d}_j||\rho_j; r_j)$.
    (d) Set $\mathsf{id}_j := [(\mathbf{r}_j, \mathsf{sn}_j, \mathsf{cm}_j)]$.
12. For each input *of type non-record* $\mathsf{i}_j \in [\mathsf{i}_j]_1^n$, do as follows.
    (a) Parse $\mathsf{i}_j$ as $(v_j, \mathsf{d}_j)$.
    (b) If $v_j$ is public, compute $c_j := \mathsf{CRH.Eval}(\mathsf{fid}, \mathsf{tvk}, j, \mathsf{d}_j)$.
    (c) Else if $v_j$ is private:
        i. Compute the input view key $\mathsf{ivk}_j := \mathsf{CRH.Eval}(\mathsf{fid}, \mathsf{tvk}, j)$.
        ii. Compute the ciphertext $c_j := \mathsf{ENC.Enc}(\mathsf{pp_{ENC}}, \mathsf{ivk}_j, \mathsf{d}_j)$.
        iii. Set $\mathsf{id}_j := [(\mathsf{i}_j, c_j)]$.
13. Output $\mathsf{req} := (\mathsf{ack}, \mathcal{P}, f, [\mathsf{id}_j]_1^n, \mathsf{rvk}, \mathsf{tvk}, \mathsf{tpk}, \mathsf{tcm}, \sigma)$.
```

The final authorization is returned in VM.Authorize, defined below.

```
$\mathsf{VM.Authorize}(M, \mathsf{ask}, \mathcal{P}, f, [\mathsf{i}_j]_1^n, \mathsf{root}) \to \mathsf{auth}$:
1. Parse $\mathcal{P}$ as $(\mathsf{pid}, \boldsymbol{f}, \mathsf{d})$.
2. Check the function membership in the program: $f \in \boldsymbol{f}$. Else, output $\perp$.
3. Parse $f$ as $(\mathsf{pid}, \mathsf{fid}, [\mathsf{ti}_j]_1^n, [\mathsf{to}_j]_1^m, \boldsymbol{\mathcal{I}}, \boldsymbol{\mathcal{F}})$.
4. Check that the inputs types $[\mathsf{ti}_j]_1^n$ are consistent with $[\mathsf{i}_j]_1^n$ as specified in the function signature.
5. Construct the request $\mathsf{req} := \mathsf{VM.ConstructRequest}(M, \mathsf{ask}, \mathcal{P}, f, [\mathsf{i}_j]_1^n)$.
6. Initialize the authorization stack with the request $\mathsf{auth} := [\mathsf{req}]$.
7. For each instruction $\mathcal{I}_k \in \boldsymbol{\mathcal{I}}$ :
    (a) If $\mathcal{I}_k$ is not a function call, execute the instruction and update the machine state $M_k$ to $M_{k+1}$.
    (b) Else, if $\mathcal{I}_k$ invokes some function $g$ in the scope of program $\mathcal{Q}$, append to $\mathsf{auth}$ the output of
        $\mathsf{VM.Authorize}(M_k, \mathsf{ask}, \mathcal{Q}, g, \{\mathsf{i}'_j\}_{j=1}^s)$, where input $\mathsf{i}'_j$ is retrieved from the machine state $M_k$.
8. If $\mathsf{root} = 1$, append to the root request the signer consistency commitment as follows:
    (a) Parse $\mathsf{auth}$ as $(\mathsf{req}_1, \ldots, \mathsf{req}_d)$.
    (b) For each $i \in \{1, \ldots, d\}$, retrieve the transition commitment $\mathsf{tcm}_i$.
    (c) Compute $\mathsf{scm} := \mathsf{CRH.Eval}(\mathsf{pp_{CRH}}, \mathsf{apk}||\mathsf{tcm}_1|| \ldots ||\mathsf{tcm}_d)$.
    (d) Update the root request: $\mathsf{req}_1 = \mathsf{req}_1 \cup [\mathsf{scm}]$.
9. Output $(M', \mathsf{auth})$.
```

### 6.3.3 Transition executions

VM.ExecuteTr produces a list of transitions by executing the requests in an authorization and an instance-witness tuple for $\mathcal{R}_{\mathsf{auth}}$. To achieve this, VM.ExecuteTr will pop the top-most request from the authorization and execute the instructions for its corresponding function sequentially. If any instruction is a function call to a different function, VM.ExecuteTr is invoked recursively. Otherwise, the machine state is updated and the instruction execution continues. An instance-witness tuple $(\mathbb{x}_{\mathsf{req}}, \mathbb{w}_{\mathsf{req}})$ for $\mathcal{R}_{\mathsf{req}}$ is created to prove the correctness of each request verification, and the tuple $(\mathbb{x}_f, \mathbb{w}_f)$ for $\mathcal{R}_f$ is created to prove the correctness of each function execution.

```
$\mathsf{VM.ExecuteTr}(M, \mathsf{tpk}, \mathsf{auth}) \to (M', \boldsymbol{T}, \mathsf{aux})$:
1. Pop the top-most request req from the authorization auth.
2. Parse auth as $[\mathsf{req}_i]_1^d$.
3. Fetch the request verification program deployment $\mathsf{dp}_{\mathsf{req}} := \mathbf{L}.\mathsf{FetchDeployment}(\mathcal{R}_{\mathsf{req}})$ and retrieve
   the circuit keys $(\mathsf{ipk}_{\mathsf{req}}, \mathsf{ivk}_{\mathsf{req}})$.
4. Construct the instance-witness tuple for $\mathcal{R}_{\mathsf{req}}$ as follows. For each $\mathsf{req} \in [\mathsf{req}_i]_1^d$:
    (a) Parse req as $(\mathsf{apk}, \mathcal{P}, f, [\mathsf{id}_i]_1^N, \mathsf{tvk}, \mathsf{scm}, \mathsf{tcm}, \sigma)$.
    (b) Parse $f$ as $(\mathsf{pid}, \mathsf{fid}, [\mathsf{ti}_j]_1^n, [\mathsf{to}_j]_1^m, \boldsymbol{\mathcal{I}}, \boldsymbol{\mathcal{F}})$.
    (c) For each input $\mathsf{id}_i \in [\mathsf{id}_i]_1^n$ that is of type record, retrieve the serial number $\mathsf{sn}_i$.
    (d) For each input $\mathsf{id}_k \in [\mathsf{id}_i]_1^n$ that is of type non-record, retrieve the plaintext or ciphertext hash $c_k$.
    (e) Set the instance $\mathbb{x}_{\mathsf{req}} := (\mathcal{P}, f, [\mathsf{sn}_i]_1^n, [c_k]_1^s, [\mathsf{i}_k^{\mathsf{pub}}]_1^{s_1}, \mathsf{tcm}, \mathsf{tpk}, \sigma, m)$.
    (f) Set the witness $\mathbb{w}_{\mathsf{req}} := (\mathsf{ack}, [\mathsf{cm}_i]_1^m, [\mathbf{r}_i]_1^m, [\mathsf{i}_j]_1^n, \mathsf{rvk}, \mathsf{tvk})$.
5. Execute the function $f$ as follows. For each instruction $\mathcal{I}_k \in \boldsymbol{\mathcal{I}}$:
    (a) If $\mathcal{I}_k$ is not a function call, execute the instruction and update the machine state $M_k$ to $M_{k+1}$.
    (b) Else, if $\mathcal{I}_k$ is a function call to $g$, compute $(\boldsymbol{T}', \mathbb{x}'_{\mathsf{req}}, \mathbb{w}'_{\mathsf{req}}) := \mathsf{VM.ExecuteTr}(M_k, \mathsf{tpk}', \mathsf{auth})$.
6. Compute the transition id: $\mathsf{tid} := \mathsf{CRH}^{\mathsf{TID}}.\mathsf{Eval}([\mathsf{i}_j]_1^n, [\mathsf{o}_j]_1^m)$.
7. Create the transition: $T := (\mathsf{tid}, \mathsf{pid}, \mathsf{fid}, [\mathsf{i}_j]_1^n, [\mathsf{o}_j]_1^m, \mathsf{tpk}, \mathsf{tcm}, \mathsf{scm})$.
8. Set the function satisfaction instance $\mathbb{x}_f := (\mathcal{P}, f, [\mathsf{sn}_i]_1^n, [\mathsf{cm}'_i]_1^m, [\mathsf{id}_i]_1^s, [\mathsf{id}'_i]_1^k, [\mathsf{i}_j]_1^n)$
9. Set the function satisfaction witness $\mathbb{w}_f := (\mathsf{ack}, [\mathbf{r}_i]_1^m, [\mathbf{r}'_j]_1^n, [\mathsf{i}_j]_1^n, [\mathsf{o}_j]_1^n, \mathsf{tvk})$.
10. Set the transitions list $\boldsymbol{T} = \boldsymbol{T}' \cup [T]$,
11. Set the authorization instance $\mathbb{x}_{\mathsf{auth}} := (\mathbb{x}_{\mathsf{req}}, \mathbb{x}'_{\mathsf{req}})$ and witness $\mathbb{w}_{\mathsf{auth}} := (\mathbb{w}_{\mathsf{req}}, \mathbb{w}'_{\mathsf{req}})$.
12. Output $(\boldsymbol{T}, \mathsf{aux} := (\mathbb{x}_{\mathsf{auth}}, \mathbb{w}_{\mathsf{auth}}))$.
```

### 6.3.4 Transaction executions

VM.ExecuteTx produces an execution transaction provided a list of transitions $\boldsymbol{T}$ and an auxiliary input aux containing an instance-witness tuple $(\mathbb{x}_{\mathsf{auth}}, \mathbb{w}_{\mathsf{auth}})$ for $\mathcal{R}_{\mathsf{auth}}$. In order to prove that non-ephemeral input records are consumed and output records are created properly, VM.ExecuteTx creates instance-witness tuples for the local inclusion relation $\mathcal{R}_{\mathsf{loc}}$ and global inclusion relation $\mathcal{R}_{\mathsf{glb}}$, and uses these, along with $(\mathbb{x}_{\mathsf{auth}}, \mathbb{w}_{\mathsf{auth}})$, to construct an instance-witness tuple for the execute relation $\mathcal{R}_{\mathsf{ex}}$. Finally, VM.ExecuteTx invokes the preprocessing argument NIZK for $\mathcal{R}_{\mathsf{ex}}$ to construct a proof $\pi$ of correct state execution.

```
$\mathsf{VM.ExecuteTx}(M, \boldsymbol{T}, \mathsf{aux}) \to (M', \mathsf{ex})$:
1. Parse the transition list $\boldsymbol{T}$ as $[T_i]_1^n$.
2. Parse the auxiliary input aux as $(\mathbb{x}_{\mathsf{auth}}, \mathbb{w}_{\mathsf{auth}})$.
3. Retrieve the current global state root: $\mathcal{G}_{\mathsf{state}} := \mathbf{L}.\mathsf{Digest}$.
4. Compute the local state root: $\ell_{\mathsf{state}} := \mathsf{CRH}^{\mathsf{TID}}.\mathsf{Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{tid}_1 || \ldots || \mathsf{tid}_n)$.
5. Fetch the program deployment for $\mathcal{R}_{\mathsf{glb}}$: $(\mathsf{ipk}_{\mathrm{G}}, \mathsf{ivk}_{\mathrm{G}}) := \mathbf{L}.\mathsf{FetchDeployment}(\mathcal{R}_{\mathsf{glb}})$.
6. Fetch the program deployment for $\mathcal{R}_{\mathsf{loc}}$: $(\mathsf{ipk}_{\mathrm{L}}, \mathsf{ivk}_{\mathrm{L}}) := \mathbf{L}.\mathsf{FetchDeployment}(\mathcal{R}_{\mathsf{loc}})$.
7. Compute the instance-witness tuples for $\mathcal{R}_{\mathsf{glb}}$ as follows. For each transition $T_i \in \boldsymbol{T}$:
    (a) Parse $T_i$ as $(\mathsf{tid}, \mathsf{pid}, \mathsf{fid}, [\mathsf{i}_j]_1^n, [\mathsf{o}_j]_1^m, \mathsf{tpk}, \mathsf{tcm}, \mathsf{scm})$.
    (b) Retrieve the non-ephemeral records appearing in the function inputs $[\mathbf{r}_i]_1^m \subset [\mathsf{i}_j]_1^n$.
    (c) For each $\mathbf{r}_i \in [\mathbf{r}_i]_1^m$, compute the commitment $\mathsf{cm}_i := \mathsf{CM.Commit}(\mathsf{pp_{CM}}, v_j||\mathsf{apk}||\mathsf{d}_j||\rho_j; r_j)$.
    (d) For each $\mathbf{r}_i \in [\mathbf{r}_i]_1^m$, compute the ledger authentication path $\mathbb{w}_{\mathbf{L},i} := \mathbf{L}.\mathsf{Prove}(\mathsf{cm}_i)$.
    (e) Construct the global inclusion instance $\mathbb{x}_{\mathrm{G},i}$ for $\mathcal{R}_{\mathsf{glb}}$ as $\mathbb{x}_{\mathrm{G},i} := (\mathcal{G}_{\mathsf{state}}, [\mathsf{sn}_i]_{i=1}^m)$.
    (f) Construct the global inclusion witness $\mathbb{w}_{\mathrm{G},i}$ for $\mathcal{R}_{\mathsf{glb}}$ as $\mathbb{w}_{\mathrm{G},i} := ([\mathbf{r}_i]_1^m, [\mathbb{w}_{\mathbf{L},i}]_1^m)$.
 8. Set the grand global inclusion instance $\mathbb{x}_{\mathrm{G}}$ for $\mathcal{R}_{\mathsf{glb}}$ as $\mathbb{x}_{\mathrm{G}} := (\mathbb{x}_{\mathrm{G},1}, \ldots, \mathbb{x}_{\mathrm{G},n})$.
 9. Set the grand global inclusion witness $\mathbb{w}_{\mathrm{G}}$ for $\mathcal{R}_{\mathsf{glb}}$ as $\mathbb{w}_{\mathrm{G}} := (\mathbb{w}_{\mathrm{G},1}, \ldots, \mathbb{w}_{\mathrm{G},n})$.
10. Compute the instance-witness tuples for $\mathcal{R}_{\mathsf{loc}}$ as follows. For each transition $T_i \in \boldsymbol{T}$:
    (a) Parse $T_i$ as $(\mathsf{tid}, \mathsf{pid}, \mathsf{fid}, [\mathsf{i}_j]_1^n, [\mathsf{o}_j]_1^m, \mathsf{tpk}, \mathsf{tcm}, \mathsf{scm})$.
    (b) Retrieve the records appearing in the function inputs and outputs $[\mathbf{r}_i]_1^m \subset [\mathsf{i}_j]_1^n \cup [\mathsf{o}_j]_1^m$.
    (c) For each $\mathbf{r}_i \in [\mathbf{r}_i]_1^m$, compute the commitment $\mathsf{cm}_i := \mathsf{CM.Commit}(\mathsf{pp_{CM}}, v_j||\mathsf{apk}||\mathsf{d}_j||\rho_j; r_j)$.
    (d) For each $\mathbf{r}_i \in [\mathbf{r}_i]_1^m$, compute the execution authentication path $\mathbb{w}_{\mathbf{L},i} := \mathbf{T}.\mathsf{Prove}(\ell_{\mathsf{state}}, \mathsf{cm}_i)$.
    (e) Construct the local inclusion instance $\mathbb{x}_{\mathrm{L},i}$ for $\mathcal{R}_{\mathsf{loc}}$ as $\mathbb{x}_{\mathrm{L},i} := (\ell_{\mathsf{state}}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_i]_1^n, [\mathsf{id}_k]_1^s)$.
    (f) Construct the local inclusion witness $\mathbb{w}_{\mathrm{L},i}$ for $\mathcal{R}_{\mathsf{loc}}$ as $\mathbb{w}_{\mathrm{L},i} := ([\mathbf{r}_i]_1^m, [\mathbb{w}_{\mathbf{L},i}]_1^m, [\mathbf{r}_i]_1^m, [\mathbb{w}'_{\mathbf{L},i}]_1^m)$.
11. Set the grand local inclusion instance $\mathbb{x}_{\mathrm{L}}$ for $\mathcal{R}_{\mathsf{loc}}$ as $\mathbb{x}_{\mathrm{L}} := (\mathbb{x}_{\mathrm{L},1}, \ldots, \mathbb{x}_{\mathrm{L},n})$.
12. Set the grand local inclusion witness $\mathbb{w}_{\mathrm{L}}$ for $\mathcal{R}_{\mathsf{loc}}$ as $\mathbb{w}_{\mathrm{L}} := (\mathbb{w}_{\mathrm{L},1}, \ldots, \mathbb{w}_{\mathrm{L},n})$.
13. Set the execution instance $\mathbb{x}_{\mathsf{ex}}$ for $\mathcal{R}_{\mathsf{ex}}$: $\mathbb{x}_{\mathsf{ex}} := (\mathbb{x}_{\mathsf{auth}}, \mathbb{x}_{\mathrm{G}}, \mathbb{x}_{\mathrm{L}})$.
14. Set the execution witness $\mathbb{w}_{\mathsf{ex}}$ for $\mathcal{R}_{\mathsf{ex}}$: $\mathbb{w}_{\mathsf{ex}} := (\mathbb{w}_{\mathsf{auth}}, \mathbb{w}_{\mathrm{G}}, \mathbb{w}_{\mathrm{L}})$.
15. Fetch the program deployment for $\mathcal{R}_{\mathsf{ex}}$: $\mathsf{dp}_{\mathsf{ex}} := \mathbf{L}.\mathsf{FetchDeployment}(\mathcal{R}_{\mathsf{ex}})$.
16. Retrieve the circuit keys $(\mathsf{ipk}, \mathsf{ivk})$ from $\mathsf{dp}_{\mathsf{ex}}$.
17. Compute the proof for $\mathcal{R}_{\mathsf{ex}}$ as $\pi := \mathsf{NIZK.Prove}(\mathsf{ipk}, \mathbb{x}_{\mathsf{ex}}, \mathbb{w}_{\mathsf{ex}})$.
18. Set the execution $\mathsf{ex} := (\boldsymbol{T}, \mathbb{x}_{\mathsf{ex}}, \pi)$.
19. Output the execution $\mathsf{ex}$.
```

### 6.3.5 Finalize executions

On input the public parameters, a list of transitions $\boldsymbol{T}$, and the current machine state $M$, VM.Finalize will update global mappings maintained on the ledger as follows.

```
$\mathsf{VM.Finalize}(\mathsf{pp}, \boldsymbol{T}, M) \to M'$:
 1. For each transition $T \in \boldsymbol{T}$:
    (a) Parse the transition $T$ as $(\mathsf{tid}, \mathsf{pid}, \mathsf{fid}, [\mathsf{i}_j]_1^n, [\mathsf{o}_j]_1^m, \mathsf{tpk}, \mathsf{tcm}, \mathsf{scm})$.
    (b) Parse the function associated to fid as $(\mathsf{pid}, \mathsf{fid}, [\mathsf{ti}_j]_1^n, [\mathsf{to}_j]_1^m, \boldsymbol{\mathcal{I}}, \boldsymbol{\mathcal{F}})$.
    (c) Parse the finalize scope $\boldsymbol{\mathcal{F}}$ in terms of the finalize inputs and commands $(\boldsymbol{\mathcal{F}}_I, \boldsymbol{\mathcal{F}}_C)$.
    (d) For each finalize input $\mathsf{f} \in \boldsymbol{\mathcal{F}}_I$, load $\mathsf{f}$ into the machine state $M$.
    (e) For each finalize command $\mathcal{C}_k \in \boldsymbol{\mathcal{F}}_C$:
        i. Execute the command $\mathcal{C}_k$ and update the machine state $M_k$ to $M_{k+1}$.
        ii. Update the state of the persistent mappings on the ledger: $\mathbf{L}.\mathsf{UpdateMappings}(M_{k+1})$.
 2. Output the updated machine state $M^{|\boldsymbol{\mathcal{F}}_C|+1}$.
```

### 6.3.6 Deployments

On input the public parameters pp, a program $\mathcal{P}$, and machine state $M$, VM.Synthesize synthesizes the proving and verifying keys corresponding to the NP relation $\mathcal{R}_f$ for each $f \in \mathcal{P}$. Since NIZK is a universal preprocessing argument, the circuit keys for $\mathcal{R}_f$ can be computed deterministically from $\mathsf{NIZK.Specialize}$ provided the universal parameters $\mathsf{pp_U}$.

```
$\mathsf{VM.Synthesize}(\mathsf{pp}, \mathcal{P}, M) \to (M', \mathsf{dp})$:
1. Parse $\mathcal{P}$ as $(\mathsf{pid}, \boldsymbol{f}, \mathsf{d})$.
2. Retrieve the universal parameters $\mathsf{pp_U}$ from $\mathsf{pp}$.
3. Initialize $\boldsymbol{M} := [\,]$.
4. For each $f_i \in \boldsymbol{f}$:
    (a) Sample the circuit-specific keys $(\mathsf{ipk}_i, \mathsf{ivk}_i) \leftarrow \mathsf{NIZK.Specialize}(\mathsf{pp_U}, \mathcal{R}_{f_i})$.
    (b) Update $\boldsymbol{M} = \boldsymbol{M} \cup (f_i, (\mathsf{ipk}_i, \mathsf{ivk}_i))$.
5. Output $\mathsf{dp} := (\mathcal{P}, \boldsymbol{M})$.
```

### 6.3.7 Execution Verifications

On input query access to the ledger $\mathbf{L}$, machine state $M$, and the execution $\mathsf{ex}$, $\mathsf{VM.VfyExec}$, outputs the updated machine state $M'$ and a decision bit $b$ indicating that $\mathsf{ex}$ is consistent with respect to $\mathcal{R}_{\mathsf{ex}}$.

```
$\mathsf{VM.VfyExec}^{\mathbf{L}}(M, \mathsf{ex}) \to (M', b)$.
1. Parse the execution ex as $(\boldsymbol{T}, \mathbb{x}_{\mathsf{ex}}, \pi)$.
2. Fetch the program deployment for $\mathcal{R}_{\mathsf{ex}}$: $\mathsf{dp_{ex}} := \mathbf{L}.\mathsf{FetchDeployment}(\mathcal{R}_{\mathsf{ex}})$.
3. Retrieve the circuit keys $(\mathsf{ipk}, \mathsf{ivk})$ from $\mathsf{dp_{ex}}$.
4. Check that the proof $\pi$ is valid: $\mathsf{NIZK.Verify}(\mathsf{ivk}, \mathbb{x}_{\mathsf{ex}}, \pi) = 1$.
5. Output a decision bit $b$.
```

## A Appendix

### A.1 Instructions

Aleo instructions are operations which act on the *register* operand. Below we list the instructions that Aleo supports. Note that `first`, `second`, and `destination` are registers.

- Abs: Compute the absolute value of `first`, checking for overflow, and storing the outcome in `destination`.
- AbsWrapped: Compute the absolute value of `first`, wrapping around at the boundary of the type, and storing the outcome in `destination`.
- Add: Adds `first` with `second`, storing the outcome in `destination`.
- AddWrapped: Adds `first` with `second`, wrapping around at the boundary of the type, and storing the outcome in `destination`.
- And: Performs a bitwise and operation on `first` and `second`, storing the outcome in `destination`.
- AssertEq: Asserts `first` and `second` are equal.
- AssertNeq: Asserts `first` and `second` are **not** equal.
- Call: Calls a closure on the operands.
- Cast: Casts the operands into the declared type.
- CommitBHP256: Performs a BHP commitment on inputs of 256-bit chunks.
- CommitBHP512: Performs a BHP commitment on inputs of 512-bit chunks.
- CommitBHP768: Performs a BHP commitment on inputs of 768-bit chunks.
- CommitBHP1024: Performs a BHP commitment on inputs of 1024-bit chunks.
- CommitPED64: Performs a Pedersen commitment on up to a 64-bit input.
- CommitPED128: Performs a Pedersen commitment on up to a 128-bit input.
- Div: Divides `first` by `second`, storing the outcome in `destination`.
- DivWrapped: Divides `first` by `second`, wrapping around at the boundary of the type, and storing the outcome in `destination`.
- Double: Doubles `first`, storing the outcome in `destination`.
- GreaterThan: Computes whether `first` is greater than `second` as a boolean, storing the outcome in `destination`.
- GreaterThanOrEqual: Computes whether `first` is greater than or equal to `second` as a boolean, storing the outcome in `destination`.
- HashBHP256: Performs a BHP hash on inputs of 256-bit chunks.
- HashBHP512: Performs a BHP hash on inputs of 512-bit chunks.
- HashBHP768: Performs a BHP hash on inputs of 768-bit chunks.
- HashBHP1024: Performs a BHP hash on inputs of 1024-bit chunks.
- HashKeccak256: Performs a Keccak hash, outputting 256 bits.
- HashKeccak384: Performs a Keccak hash, outputting 384 bits.
- HashKeccak512: Performs a Keccak hash, outputting 512 bits.
- HashPED64: Performs a Pedersen hash on up to a 64-bit input.
- HashPED128: Performs a Pedersen hash on up to a 128-bit input.
- HashPSD2: Performs a Poseidon hash with an input rate of 2.
- HashPSD4: Performs a Poseidon hash with an input rate of 4.
- HashPSD8: Performs a Poseidon hash with an input rate of 8.
- HashSha3_256: Performs a SHA-3 hash, outputting 256 bits.
- HashSha3_384: Performs a SHA-3 hash, outputting 384 bits.
- HashSha3_512: Performs a SHA-3 hash, outputting 512 bits.

- HashManyPSD2: Performs a Poseidon hash with an input rate of 2.
- HashManyPSD4: Performs a Poseidon hash with an input rate of 4.
- HashManyPSD8: Performs a Poseidon hash with an input rate of 8.
- Inv: Computes the multiplicative inverse of `first`, storing the outcome in `destination`.
- IsEq: Computes whether `first` equals `second` as a boolean, storing the outcome in `destination`.
- IsNeq: Computes whether `first` does not equal `second` as a boolean, storing the outcome in `destination`.
- LessThan: Computes whether `first` is less than `second` as a boolean, storing the outcome in `destination`.
- LessThanOrEqual: Computes whether `first` is less than or equal to `second` as a boolean, storing the outcome in `destination`.
- Modulo: Computes `first` mod `second`, storing the outcome in `destination`.
- Mul: Multiplies `first` with `second`, storing the outcome in `destination`.
- MulWrapped: Multiplies `first` with `second`, wrapping around at the boundary of the type, and storing the outcome in `destination`.
- MulWrapped: Multiplies `first` with `second`, wrapping around at the boundary of the type, and storing the outcome in `destination`.
- Not: Performs a bitwise `not` operation on `first`, storing the outcome in `destination`.
- Or: Performs a bitwise `or` operation on `first` and `second`, storing the outcome in `destination`.
- Parse: Parses the input string according to the rules of the specified data type and stores it in `destination`.
- SignBit: Determines if the sign bit is set for `first`, storing the outcome in `destination` as a boolean.
- Sub: Subtracts `second` from `first`, storing the outcome in `destination`.
- SubWrapped: Subtracts `second` from `first`, wrapping around at the boundary of the type, and storing the outcome in `destination`.
- VerifySchnorr: Verifies a Schnorr signature given a public key, message, and signature.
- Xor: Performs a bitwise XOR operation on `first` and `second`, storing the outcome in `destination`.

## References

[BCG$^+$14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 459–474, 2014.

[BCG$^+$15] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, SP '15, pages 287–304, 2015.

[BCG$^+$18a] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. Cryptology ePrint Archive, Paper 2018/962, 2018.

[BCG$^+$18b] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. Cryptology ePrint Archive, Paper 2018/962, 2018. `https://eprint.iacr.org/2018/962`. URL: `https://eprint.iacr.org/2018/962`.

[BGG18] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-SNARK, 2018.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017.

[Bit15] Bitcoin. Some miners generating invalid blocks. `https://bitcoin.org/en/alert/2015-07-04-spv-mining`, 2015.

[BS23] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Version 0.6*, 2023.

[Eth16] Ethereum. I thikn the attacker is this miner - today he made over $50k. `https://www.reddit.com/r/ethereum/comments/55xh2w/i_thikn_the_attacker_is_this_miner_today_he_made/`, 2016.

[KMS$^+$16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, pages 839–858, 2016.

[LTKS15] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS '15, pages 706–719, 2015.

[Nak09] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2009. URL: `http://www.bitcoin.org/bitcoin.pdf`.

[Woo17] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2017. `http://yellowpaper.io`.

[XCZ$^+$22] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. VERI-ZEXE: Decentralized private computation with universal setup. Cryptology ePrint Archive, Paper 2022/802, 2022. `https://eprint.iacr.org/2022/802`. URL: `https://eprint.iacr.org/2022/802`.

[ZCa16] ZCash parameter generation. `https://z.cash/technology/paramgen.html`, 2016. Accessed: 2017-09-28.