RGB I.0
SCALABLE CONSENSUS FOR CLIENT-SIDE VALIDATED SMART CONTRACTS

July 30, 2025

DR. MAXIM ORLOVSKY

Institute for Distributed and Cognitive Systems, LNP/BP Labs & UBIDECO Labs, Lugano, Switzerland;
Pandora Prime SA, Neuchatel, Switzerland

Abstract. The paper defines a novel type of consensus for a distributed smart contract system, named RGB, which is based on the concept of client-side validation, separating the contract state and operations from the blockchain. With this approach, contracts are sharded (each contract is a standalone shard), kept, and validated only by contract participants, providing native scalability and privacy mechanisms, exceeding all existing blockchain-based smart contract systems while not compromising on security or decentralization. The system is designed to operate on top of compatible layers 1, such as an UTXO-based blockchain (e.g., Bitcoin) without relying on it for transaction ordering or state replication. Instead, RGB keeps the state client-side, operating as partially replicated state machines (PRiSM). It employs a novel SONIC (State machine with Ownership Notation Involving Capabilities) architecture, which provides capability-based access control to the contract state, individually owned and operated by a well-defined contract parties via novel single-use seal mechanism. RGB does state validation using zk-AluVM virtual machine, designed to support zk-STARK provers. It has a single security assumption of the collision-resistance hash function and, thus, is quantum-secure. The proposed RGB consensus is distinct from traditional blockchain-based smart contract systems; it is scalable, provably-secure, and formally verifiable.

1. Introduction

1.1. **Background.** The idea of smart contracts, originating from Nick Szabo [1], has been an inspiration for almost a generation of researchers and innovators. Nevertheless all existing attempts for building permissionless smart contracts were based on fully-replicable blockchain consensus, and thus were failing to solve the trilemma of being

- scalable as $O(1)$, or at least as $O(\log n)$;
- universal ("Turing-complete");
- decentralized, permissionless, and trustless.

The main reason was the fact that all the mentioned approaches require all participants to verify the state and state transition function of all world smart contracts, which, for obvious reasons, cannot scale.

Multiple attempts has being made to solve blockchain non-scalability issue via adding layers on top: sidechains [2, 3], statechains [4], state channels [5, 6], plasma [7], optimistic rollups [8], validiums [9], zk-rollups [10, 11, 12]. However, all of the approaches result in either centralization, higher trust assumptions, possibility of a censorship, or suffer from economical non-scalability [13, 14, 15], like in the case of the infamous Lightning network inbound liquidity [16, 17], routing [18] and centralization issues [19, 20]. Also, all layer 2 solutions has the state which must be eventually published to the blockchain, so, the state transition business logic also has to be global. This becomes especially an issue for blockchains designed for a "Turing-complete" contracts, serving a complete VM, like in Ethereum [21] or Cardano [22]. Thus, all of them may save only a constant factor and cannot solve the fundamental limitation created by a blockchain-based consensus.

1.2. **Client-side Validation.** This problem can be addressed with *client-side validation* – the paradigm proposed by Peter Todd in 2016 [23]. Its core idea is that state validation in a distributed system does not need to be performed globally by all parties of a decentralized protocol; instead, only parties involved in (or affected by) a specific state transition need to perform the validation. With this approach it is sufficient to publish only a cryptographic commitment, with no state data, and the parties of the specific transaction be able to verify that it is correct, final and singular — without the rest of the world even knowing of it. This is achievable with a new class of cryptographic protocols, also proposed by Peter Todd, named single-use seals [24, 25, 26, 27], which allow proving both finality and singularity[1]of a state transition in a noninteractive way.

Based on the ideas of Peter Todd, Giacomo Zucco had proposed an early idea of RGB as a client-side validated asset system, leveraging Bitcoin transactions (both on-chain and off-chain, like in Lightning network) for the *single-use seals* protocol [29]. However, this early idea and prototype implementation was lacking programmability, support for non-trivial state, ability for a Bitcoin UTXO to be used by multiple assets, and was not zk-friendly.

1.3. **Our Contribution.** In our work, we have developed the original idea of RGB into the first universal client-side validated distributed smart contracting system, solving all the above-mentioned issues. RGB employs a novel SONIC (State machine with Ownership Notation Involving Capabilities) architecture, leveraging zk-AluVM virtual machine [30] and zk-STARKs [31] for scalability, security, and formal verification. The protocol operates as partially

[1]By *singularity* here we mean a property that any atom of a contract state can be updated once and only once, in a non-concurrent way. In case of an UTXO model used in Bitcoin-type blockchains [28] this is also named an *anti-doublespending* property, but in more generic context of UTXO-based smart contracts we use the term *singularity* of state updates.

replicated state machines (PRiSM); it uses polynomial computation and capability-based access control, which makes it distinct from traditional smart contract systems. The protocol operates on top of a UTXO-based blockchain [28] (so-called *layer 1*) using it as a finality gadget; however, the blockchain consensus is not used for ordering of transactions or storing a state, unlike in all other smart contract systems. Thus, *layer 1*, in strict computer science terms, does not serve as an RGB consensus by itself, since it does not fulfill any of the properties defining a distributing consensus protocol [32]:

- ordering of transactions (RGB transactions are not published or broadcast via blockchain);
- state machine replication (no state is replicated via *layer 1*);
- atomic broadcasts (since the ordering of *layer 1* transactions does not matter for RGB).

Instead, RGB runs its own *client-side validated consensus*, where blockchain consensus acts as one of the components: the *single-use seal* medium, or *layer 1* [33]. The consensus is quantum-secure (as long as the underlying layer 1 network is quantum-secure), being based on cryptographic hash functions and not using any elliptic-curve cryptography. The consensus does not rely on any security assumptions other than the collision resistance of the selected cryptographic hash function.

This paper provides a complete formal description of the first version of the RGB consensus released to the production, named **RGB I.0**.

2. Used Notations

We use standard mathematical notation, including operators of set theory.

2.1. **Data types.** Data types are defined using standard sets, such as natural numbers $\mathbb{N}$, integers $\mathbb{Z}$, nonzero natural numbers $\mathbb{N}^+ \triangleq \mathbb{N} \setminus \{0\}$ and finite field $\mathbb{F}_q$ (where q is the order of the finite field). The bit dimensions (except for the finite field elements) are given as a subscript, like $\mathbb{N}_{256}$, with $\mathbb{N}_8$ used for both 8-bit unsigned integers and bytes.

Binary strings (scalar arrays) of fixed size are given using power notation, for instance, $\mathbb{N}_8^{32}$ is a 32-byte string[2].

Named data types are represented with capital Latin letters using LaTeX $\mathbb{MATHBB}$ font style. For example, the Boolean type is defined as a set $\mathbb{B} \triangleq \{0, 1\}$.

Scalar variables are given with Latin and Greek lowercase letters. The basis for nondecimal scalar literals is given in subscript, for example, 10_2 equals 2 in decimal notation. Decimal literals omit the basis subscript.

2.1.1. *Product types.* Tuples (ordered fixed-size set of objects of different types) are shown with angular brackets $\langle \ldots \rangle$, which contains a list of the tuple fields.

Immutable tuple fields are specified with small Latin serif letters (such as a, x) or capital Greek letters (Θ).

Mutable tuples that may change over time (tuple variables) are denoted with capital Latin serif letters (like C). To represent a sequence or individual tuple values which evolve over time (like state objects), with individual values being a lowercase indexed version of the same letter ($\mathsf{c}_0, \mathsf{c}_i$, etc.)

Tuple fields may be named or unnamed. To represent a named field of a tuple we use subscription made with serif font, such as in $\mathsf{c}_i \triangleq \langle \mathsf{C}_{\mathsf{Id}}, \ \ldots \rangle$.

2.1.2. *Sum types.* Enums (sum types) are represented as sets, and enum variables as values picked from a set of allowed values. Optional values thus belong to the set $\{\varnothing, x\}$.

Enums with no associated data are tagged with names, and a $\mathbb{N}_8$ value used in data serialization, is given as a subscript; for example $\{\mathsf{false}|_{=0}, \mathsf{true}|_{=1}\}$.

Enum values with associated data display variants or literals as variables of the corresponding type; they can include tagged names as a part of the subscript: $\{x|_{\mathsf{variantA}=0}, y|_{=1}\}$.

2.1.3. *Collections.* Collections (variable-size sets of objects of the same type, ordered or unordered) are denoted with Latin uppercase calligraphic letters (like $\mathcal{S}$). For representing set elements, we use braces $\{\ldots\}$ and the set builder notation [34]. Ordered collections are indexed with subscripts: $s_i \in \mathcal{S}$.

The type of variable-sized sets is given as an element type with a power component specifying the range of cardinality allowed for the set. For example, $\mathbb{N}_8^{[0,2^{32})}$ indicates a sequence of bytes with cardinality (length) in the range of 0 to $2^{32} - 1$.

Set definitions use a subscript to specify the ordering: $\{\cdots\}_{\prec}$ for partially ordered sets; $\{\cdots\}_{\preceq}$ for totally ordered sets; no index for unordered sets.

UTF-8 Unicode strings are represented as a sequence made up of $\mathbb{U}$ set elements; ASCII strings as a sequence made up of $\mathbb{S}$ set elements. If there are constraints on the type of ASCII characters that can be used in a string, the constraint is either given verbally as an index (for example, $\mathbb{S}_{\mathsf{printable}}$), or as a wildcard $\mathbb{S}_*$ with detailed explanation of the constraints given in the text.

Maps are written as a surjective function, mapping each element of a set of keys to a value: $M : \{\forall k \in \mathcal{K} \to v\}$. The size of the key set may be specified as a power: $M : \{\forall k \in \mathcal{K} \to v\}^{[0,10)}$.

2.2. **Function types.** Functions with a well-defined algorithm are denoted by small-case Latin serif names, followed by a colon, function domain, arrow, and function codomain, such as $\mathsf{evaluate} : \mathcal{A} \to \mathcal{B}$. The application of the function to the arguments is written as $\mathsf{evaluate}(a_i)$. Variables having function type are written as small Greek letters in italics, ϕ, and their application as $\phi(x)$. Functions depending on their previous values are defined using $\mapsto$.

2.3. **Operators.** The order of the elements in a set is indicated by $\succ$ and $\prec$. New terms are defined by $\triangleq$.

Bitwise binary operations are specified with & (AND), | (OR), $\nleftrightarrow$ (XOR) and $\sim$ (NOT)

Logical conditions are specified with $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT). Operator $\stackrel{?}{=}$ performs equivalence test and results in a Boolean value.

Operator $\Longleftrightarrow$ is used for *if and only if* condition, $\Rightarrow$ for *than*, $\nRightarrow$ for *else*, $\perp$ for program termination.

[2]The difference between $\mathbb{N}_8^{32}$ and $\mathbb{N}_{256}$ lies in the fact that the former is a byte string with a fixed encoding, while the latter is a natural number, which may be serialized either little-endian or a big-endian. For little-endian serialization $\mathbb{N}_{256}|_{\mathsf{littleEndian}} \equiv \mathbb{N}_8^{32}$. RGB consensus uses only little-endian serialization, and thus the two representations are equivalent

3. Protocol Overview

In technical terms, RGB operates as *partially replicated state machines* (PRiSM), which uses *polynomial computer architecture* SONIC (State machine with Ownership Notation Involving Capabilities).

Partially replicated means that the state is not replicated as a whole in all instances of the state machine; instead, only a part of the state required by each of the instances is propagated.

Polynomial computer means that the trace of the operations can be arithmetized to polynomials, making it possible to use any zk-STARK prover for zero-knowledge compression.

Ownership notation involving capabilities means that some types of state in RGB are assigned to specific parties (actors), and this assignment is made using *capabilities*, which are implemented using a single-use seal cryptographic scheme.

The principal components of the RGB consensus design are the following:

- SONIC polynomial computer (its consensus-related layer named "UltraSONIC") with capability-based memory and zk-AluVM virtual machine used for contract state evaluation/validation;
- RGB contracts with their state, operations and state transitions;
- single-use seals, for providing capabilities, control of state transition singularity and the finality;
- commitment schemes, using cryptographic hash functions.

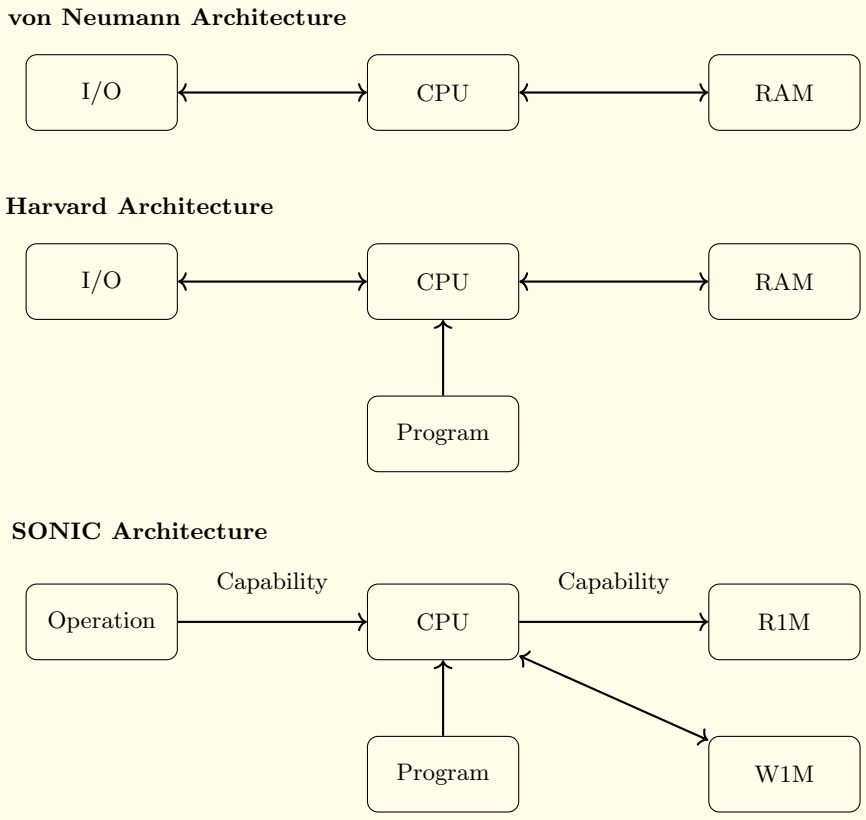
The version of RGB consensus described in this document is named **RGB-I.0**. You can find more about RGB version numbering in the RGB-6 standard [35].

4. SONIC Architecture

Most of the modern computers use von Neumann [36], Harvard, and modified Harvard architecture [37]. While offering a convenient user experience, these architectures were not designed neither for arithmetization, required in creation of zk-STARK proofs, nor for distributed systems, nor for security and formal verification. For example, random-memory access has been the source of most hacks and security breaches for more than 50 years [38, 39, 40].

Although it is possible to adopt von Neumann-style or Harvard architecture with RAM for zk-STARK provers (one may refer to Cairo [11]), this results in large proof size, high computational resource demand for producing zk-proofs, and no guaranteed ability to perform termination and other forms of formal analysis of an arbitrary program. Instead of following such an approach, RGB is using a different architecture, named SONIC (see Figure 1), which is specifically designed for formal verification, UTXO-based models, and security.

Figure 1. Comparison of different computer architectures



The architecture uses zk-AluVM virtual machine, immutable memory cells of two types (read-once memory, R1M; and write-once memory, W1M), and capability-based R1M access.

4.1. **zk-AluVM virtual CPU.** zk-AluVM is a pure functional RISC virtual machine designed for deterministic portable computing tasks [41]. It is built using AluVM ("algorithmic logic unit VM") [30, 42], a modular framework for creating RISC instruction set architectures (ISA) and registry-based virtual machines, based on category theory.

4.1.1. *Instruction Set Architecture.* zk-AluVM extends the AluVM execution control flow ISA (see Table 5) with **GFA256** ISA for arithmetic operations with finite field elements (see Table 6). It is further extended by **USONIC** ISA for accessing operation data and memory (see Table 7). The complete specification on the instruction set architecture is given in Appendix A.

4.1.2. *Control Registers.* The control registers are part of the base AluVM ISA. They are used for the program execution workflow and do not participate as explicit arguments for the instructions; however, their value may be accessed or changed by some of the instructions, as specified in Appendix A.

- **CK:** Check register with values taken from $\{\mathsf{ok}|_{=0}, \mathsf{failire}|_{=1}\}$. On a failure (like accessing register in `None` state, zero division, etc.) the register is set to the failed state. Can be reset if **CH** is `false`.
- **CH:** Halting register holding a Boolean value. Indicates whether a program must halt the moment **CK** is set to the failed state for the first time.
- **CO:** Test register, which acts as a Boolean test result. Its value is checked by branching and some halting instructions.
- **CA:** Complexity accumulator / counter, a $\mathbb{N}_{64}$ number. Each instruction has a *computational complexity measure*. This register sums up the complexity of the instructions executed.
- **CL:** Complexity limit, as an optional value $\{\varnothing, \mathbb{N}_{64}\}$. If the register has a value, once **CA** reaches or exceeds it, the VM will set **CK** to the failed state.
- **CS:** Call stack, $\{\mathbb{E}\}^{[0,256)}$, listing call entry points (see the definition of the entry point in Section 4.3 and equation 8).

4.1.3. ***GFA256*** *Registers.* The **GFA256** ISA defines **FQ** constant (read-only) register, which contains the order q of the used finite field $\mathbb{F}_q$; and 16 general registers to store finite field element values, from **E1** to **E8**, and from **EA** to **EH**. All registers are equivalent in their functionality and can be used as arguments in the instructions shown in Table 6.

4.1.4. ***USONIC*** *Registers.* The **USONIC** ISA extension defines four $\mathbb{N}_{16}$ registers, **UI1-UI4**, which acts as operation input and output iterators. They are initialized with zero, and increment on each data load instruction, such that destructible and immutable operation inputs and outputs may be iterated one after another.

For the details on how individual operations work with these registers please refer to the table 7.

4.2. **Memory.** The SONIC architecture supports two types of addressed memory:

- **Read-once memory:** (R1M), or *destructible memory* $\mathcal{D}$, having capability-based access and used for storing *owned state* accessible only by a valid actor providing a 256-bit *authentication token*;
- **Write-once memory:** (W1M), or *immutable memory* $\mathcal{I}$, which can be accessed by any party and is used to define a *global contract state*.

Both types of memory are made of addressable memory cells. An address consists of a 256-bit id of an operation and 16-bit output number which creates the memory cell:

$$\mathbb{A} \triangleq \mathbb{N}_8^{32} \times \mathbb{N}_{16} \tag{1}$$

$$\mathsf{r} \triangleq \langle \mathsf{Id}(c_i), j \rangle \in \mathbb{A} \tag{2}$$

Memory data are based on *composed value* data type, which is an ordered sequence from zero to four finite field elements:

$$\mathbb{V} \triangleq \bigcup_{n=0}^{4} \mathbb{F}_q^n \tag{3}$$

4.2.1. *R1M Memory. Read-once memory* cell consists of a single *composed value* σ, an *authentication token* α, and an optional *locking condition* $\mathcal{L}$:

$$\mathbb{D} \triangleq \mathbb{V} \times \mathbb{F}_q \times \{\varnothing, \langle \mathbb{V}, \{\varnothing, \mathbb{E}\}\rangle\} \tag{4}$$

$$\forall \mathsf{d}_i \in \mathcal{D}(\mathsf{a}) : \mathsf{d}_i \triangleq \langle \sigma, \alpha, \mathcal{L} \rangle \in \mathbb{D} \tag{5}$$

A locking condition, when present as $\langle \Theta, \ell \rangle \in \mathcal{L} \setminus \{\varnothing\}$, consists of an auxiliary data Θ and an optional entry point $\ell \in \mathbb{E}$ to a verification program (for the definition of $\mathbb{E}$ please refer to (8)).

The locking script, when present as $\varepsilon \in \ell \setminus \{\varnothing\}$, is an entry point into the AluVM program (see next section), which should succeed on execution in order for the cell to be read and destroyed.

The auxiliary data Θ may be used by a codex verifier script or the custom locking script ε to check the satisfaction of the lock conditions with an input-specific witness Ω, which will come from the spending operation input (see Section 5.3.4). For example, Θ may be a value of a public key, and the *input-specific witness* Ω will have a signature, while the script ε will validate the signature for that key.

Destructible memory cells are destroyed on the first read and cannot be accessed or referenced anymore.

4.2.2. *W1M Memory. Immutable memory* cells consist of a single *composed value* ς, and an optional binary raw data $\mathcal{R}$, up to 2^{16} bytes:

$$\mathbb{I} \triangleq \mathbb{V} \times \{\varnothing, \mathbb{N}_8^{[0;2^{16})}\}_{\preceq} \tag{6}$$

$$\forall \mathsf{e}_i \in \mathcal{I}(\mathsf{a}) : \mathsf{e}_i \triangleq \langle \varsigma, \mathcal{R} \rangle \in \mathbb{I} \tag{7}$$

The raw data do not participate in the validation process and thus are never arithmetized. Their purpose is to provide more contextual information for the user interface; they are parsed and processed using ABI rules and the standard library code (see the RGB-1010 standard [43]).

4.3. **Program.** AluVM operates as a Turing machine, running on a "tape" of a SONIC program, with the following modifications:

- the machine can't change the cells on the tape (the program is read-only);
- it has access to external data outside of the tape:
  - current contract operation (see Section 5.3),
  - addressable memory (see Section 4.2);
- its execution is bounded by *complexity measure*; each execution of an instruction increase complexity counting register **CA** in AluVM virtual CPU, and when the value exceeds the limit, provided as a part of a contract Codex (see Section 5.1), it halts, ending in a failed state.

The machine is a single-threaded and executes a single instruction at each step. The complete list of all instructions and their encodings, as well as information about registers (i.e., instruction set architecture), is given in Appendix A.

AluVM comes with a set of special-purpose control registers, described in the Section 4.1.2. Three of these registers influence the halting conditions of the machine: **CK**, **CH** and **CL**.

The program halts on the following conditions:

(1) On an unconditional halting instruction execution (see table of all instructions);
(2) On a conditional halting instruction execution *if* the value of instruction-specific register (**CO** or **CK**) is set;
(3) On any invalid operation (division by zero, accessing non-existing value or memory cell, etc.) *if* the **CH** is set;
(4) On a jump to an unknown location (unknown library id or an offset outside the bounds of the library code segment);
(5) If number of nested calls using **CS** exceeds 2^{16} (call stack overflow);
(6) Once the complexity limit, given in **CL**, is exceeded, *if* the **CL** register contains a value, independently from whether **CH** is set or not.

The complexity limit **CL** and halting flag **CH** are initialized using contract parameters from the contract codex (see Section 5.1). When a complexity limit and halting flag are set, an AluVM program is guaranteed to halt; this enables the termination analysis and simplifies program formal verification.

Execution of a program results in an execution trace consisting of instructions, input values (values in registers before instruction was executed), output values (new values in registers once the instruction is executed) and hidden parameters, coming from the external data, if they were accessed. The execution trace may be encoded using elements of the finite field $\mathbb{F}_q$ and fed to a zk-STARK prover [31]. The specific details of the encoding and selection of the zk-STARK prover are beyond the scope of this document and are a subject of future work.

To run a program, AluVM must be provided with an unordered set of known *libraries*, used by the program, and an *entry point*

$$\mathbb{E} \triangleq \mathbb{N}_8^{32} \times \mathbb{N}_{16} \tag{8}$$

$$\mathsf{h} \triangleq \langle \mathsf{Id}_{\mathsf{lib}}, p \rangle \in \mathbb{E} \tag{9}$$

consisting of a library id $\mathsf{Id}_{\mathsf{lib}}$ and an offset p in the library code segment. The library id represents a tagged hash of the library code and data. More details on library ids can be found in the AluVM documentation [42].

The number of known libraries is unbounded; therefore, the complexity limit and the call stack size are the only bounding conditions for a program. For the complete information about the AluVM library, its structure, constraints, etc., one may refer to the documentation [30].

5. Contracts

The contract is an instance of the RGB protocol. RGB consensus operates on the contract level, and does not include (as of version I.0) any cross-contract functionality[3].

Since RGB operates as a partially-replicated state machine (PRiSM), each party has a partial view over contracts, named *local contract*. A local contract is defined as

$$\mathsf{C} \triangleq \langle \Theta, \mathcal{O} \setminus \{\mathsf{c}_0\} \rangle \tag{10}$$

where Θ is a contract issue and $\mathcal{O}$ is a locally known part of the contract operations (excluding the genesis operation c_0 already present in Θ, see below).

The *issue* defines the *unique* and *global* properties of the contract; it must be known to all parties (i.e., present in each *local contract* instance) and is represented by a tuple

$$\Theta \triangleq \langle v, \mathsf{m}, \mathsf{k}, \mathsf{c}_0 \rangle \tag{11}$$

which components are defined in Table 1.

Table 1. Specification of symbols used in the RGB contract

Symbol	Type	Value range	Meaning
v	$\mathbb{N}_8$	constant 0	RGB issue data structure version
m	$\langle \mathbb{B}, \mathbb{N}_8, \mathbb{N}_{64}, \mathbb{B}^{48}, \mathbb{S}_*^{[1,100)?}, \mathbb{S}_{\mathsf{printable}}^{[1,4096)} \rangle$	n/a	Contract metadata
k	See Section 5.1	n/a	Codex (see Section 5.1)
c_0	See Section 5.3	n/a	Genesis operation

[3]However, one may still achieve cross-contract interaction outside of the consensus layer, for instance by using atomic swaps.

The commitment to the *issue* data represents a unique and global *contract id* $\mathsf{Id}(\mathsf{C})$, or C_{Id} (see Section 5 and C.2).

The contract metadata m define contract-specific parameters, which include (in the order of their position in the tuple):

(1) boolean, indicating whether the contract is a test contract,
(2) a byte representing specific *layer 1* used by the contract (see Table 2 for the list of possible values),

Table 2. List of supported Layer 1's

m_2 value	Layer 1
0_{16}	No layer 1 is used
10_{16}	Bitcoin blockchain [44]
11_{16}	Liquid sidechain [45]
20_{16}	Prime [46]

(3) ISO 8601 timestamp (a $\mathbb{Z}_{64}$ integer) of the moment the contract is issued,
(4) a 32-bit set of feature flags, $\mathbb{B}^{32}$, encoded as $\mathbb{N}_{32}$, which must be set to zero for RGB-I.0,
(5) optional name of the contract, which must start with a capital letter or a _ symbol, and may contain up to 99 ASCII letters, numbers, or _ symbol,
(6) an identity string of the contract issuer, made of ASCII printable characters.

5.1. **Codex.** The codex is a set of parameters and rules that define the contract business logic but do not define any form of a state. The *contract business logic* does not imply the way the state transitions are created; instead, it defines how an arbitrary contract operations, created with any possible rules, gets validated. If the operations pass the validation, their business logic is valid; otherwise, operations are considered invalid, and contract state transition does not happen. This paves the way to huge scalability as well as much more compact zk-STARK proofs; since any zk-proof just proves a result of a computation, not specifying the exact way of performing the computation itself. The mistake of blockchain developers was to put the actual contract business logic into the blockchain: an approach which does not scale. Client-side validation, implemented in RGB, fixes that.

Thus, the RGB codex defines an operation *validation* functions which are differentiated by the operation type: a single contract may have multiple forms of operations, which can be seen as mutating methods of the contract. These operations, when successfully validated, result in the contract state transitions.

Next, the codex defines the following contract parameters:

- Specific finite field $\mathbb{F}_q$ defined by its order q, and a bit dimension for the finite field type variables, denoted hereinafter as $\|q\|_{\mathsf{bits}}$;
- cryptographic hash function used in commitment schemes;
- specific blockchain, used as *layer 1*,
- specific single-use seal protocol, (which also defines the subset of specific blockchain networks and commitment schemes, see Table 2);

More formally, the codex is a tuple

$$\mathsf{k} \triangleq \langle v, n, d, t, f, q, \Gamma, \Lambda, V \rangle \tag{12}$$

which components are defined in Table 3.

Table 3. Specification of symbols used in the RGB codex

Symbol	Type	Required value	Meaning
v	$\mathbb{N}_8$	$= 0$	RGB codex data structure version
n	$\{\mathbb{S}\}^{[0,255]}$		Codex name, parsed as a Unicode UTF-8 string
t	$\mathbb{Z}_{64}$		ISO 8601 timestamp of the codex creation
f	$\mathbb{B}^{32}$	$= 0$	Feature flags, encoded as $\mathbb{N}_{32}$ (must be set to zero in RGB-I.0)
q	$\mathbb{Z}_q$	$= q$	Finite field order
Γ	$\langle \mathbb{B}, \mathbb{B}, \mathbb{N}_{64} \rangle$		zk-AluVM configuration for state transition verification
Λ	$\langle \mathbb{B}, \mathbb{B}, \mathbb{N}_{64} \rangle$		zk-AluVM configuration for memory access lock verification
V	$\mathbb{N}_{16} \to \langle \mathbb{N}_8^{32}, \mathbb{N}_{16} \rangle$		Entry points for verification functions using known zk-AluVM libs

Configurations for a zk-AluVM are 3-tuples, which values correspond to:

(1) Boolean flag, encoded as a byte, indicating whether a VM must halt on the first occurrence of a failure;
(2) Boolean, indicating whether a complexity limit is set;
(3) 64-bit natural number representing the complexity limit (if the pt. 2 above is set).

For details on complexity limits, please refer to AluVM documentation [30].

A codex may be used by multiple contracts – in the same way as a class, defined with an OOP programming language, may instantiate multiple objects. It is important to note that multiple contracts may reuse the same codex. In this way, there appears a natural differentiation between contract issuers and codex developers, with the former being specializing in financial services, assets, etc; and the second being specialists in computer science.

5.2. **Contract State.** A *local contract state* is fully defined by a set of its operations, $\mathcal{O}$.

Since RGB consensus needs to operate as a polynomial computer with a computation trace that can be arithmetized into a set of polynomial constraints, the state of a

contract at the level of consensus must always be represented by a mathematical construct made of elements of the finite field $\mathbb{Z}_q$, provided in the contract codex. This state is not human-readable and must be processed using specific ABIs and interfaces to be understood by humans; however, this part is outside the consensus definition, belonging to the RGB standard library, defined in the RGB-1010 standard [43].

Local contract state is a tuple $\langle \mathcal{D}, \mathcal{I} \rangle$, consisting of destructible $\mathcal{D}$ and immutable $\mathcal{I}$ memory cells, as described in Section 4.2.

Destructible state: is represented by the *read-once memory* (R1M), which is defined by contract operation destructible outputs. The atoms of the state (memory cells) can be accessed by a contract operation referencing them as inputs only once, getting destroyed by the fact of the reference.

Immutable state: is represented by the *write-once memory* (W1M, also write-once multiple-access memory), which is defined by contract operation immutable outputs and is accessed by contract operations immutable inputs.

The state of the memory is defined as a result of executing the $\mathsf{evaluate} : \mathcal{O} \to \langle \mathcal{D}, \mathcal{I} \rangle$ procedure, as described in Section 5.6 and equation (19).

5.3. **Contract Operation.** Contract operation is a tuple

$$o_i \triangleq \left\langle \mathsf{c}_i, \mathcal{S}_i, \{\varnothing, w_i\} \right\rangle \tag{13}$$

consisting of:

- client-side information of contract state change, represented by a tuple c_i;
- an unordered map of single-use seal definitions performed by an operation,

$$\mathcal{S}(x) : \{y_x \in \mathcal{Y} \to s_x\} \tag{14}$$

  where s_x is a seal definition;
- a *seal close witness*, which values must belong to a set of either unit value, or a specific witness w_i.

5.3.1. *Client-side Operation.* A client-side part of the operation is represented by a tuple

$$\mathsf{c}_i \triangleq \langle v, \mathsf{C_{Id}}, \phi, \lambda, \Upsilon, \mathcal{A}, \mathcal{B}, \mathcal{Y}, \mathcal{Z} \rangle \tag{15}$$

which meaning is given in the table 4.

5.3.2. *Genesis Operation.* The contract *genesis* is a special type of operation, which contains no inputs, and where $\mathsf{C_{Id}}$ is replaced with $\mathsf{k_{Id}}$:

$$\mathsf{c}_0 \triangleq \langle \pi, \mathsf{k_{Id}}, \phi, \lambda, \Upsilon, \varnothing, \varnothing, \mathcal{Y}, \mathcal{Z} \rangle \tag{16}$$

5.3.3. *Operation Id.* Each operation is identified by an operation id, $\mathsf{Id}(c_i)$, which is computed by hashing serialized data for the client-side part of the operation, as described in Section 6 and Appendix C.4. For genesis, the value of $\mathsf{k_{Id}}$ is replaced by the value of $\mathsf{C_{Id}}$ before the id of the operation is calculated (see C.3).

5.3.4. *Operation Destructible Inputs.* Operations, which are not *genesis* and not *state extensions*, have a non-empty destructible memory cell refs, which are composed of tuples

$$a_k \in \mathcal{A}_i \triangleq \langle \mathsf{d}_x, \Omega \rangle \tag{17}$$

where d_x represents a reference to a previously-defined destructible memory cell, and Ω is an *input-specific witness* (not to mix with single-use *seal close witness* w_i, and *operation-level witness* Υ). The *input-specific witness* Ω participates in the capability-based memory access, as it is described in the Section 4.2.1 above.

Table 4. Specification of symbols used in the RGB operation

Symbol	Type	Meaning
v	$\mathbb{N}_8$	RGB consensus version (must be set to 0)
$\mathsf{C_{Id}}$	$\mathbb{N}_8^{32}$	Contract id
ϕ	$\mathbb{N}_{16}$	Call id: an index of a verification entry point V from the codex k, used to verify the transition
λ	$\mathbb{N}_{16}$	Nonce (used for creating vanity contract ids)
Υ	$\mathbb{V}$	Operation-level witness data (unrelated to single-use seal witness!)
$\mathcal{A}$	$\{\mathbb{A} \times \mathbb{V}\}_{\preceq}^{[0,2^{16})}$	Destructible memory refs (input)
$\mathcal{B}$	$\{\mathbb{A}\}_{\preceq}^{[0,2^{16})}$	Immutable memory refs (input)
$\mathcal{Y}$	$\{\mathbb{D}\}_{\preceq}^{[0,2^{16})}$	Destructible memory declaration (output)
$\mathcal{Z}$	$\{\mathbb{I}\}_{\preceq}^{[0,2^{16})}$	Immutable memory declaration (output)

5.4. **Single-use Seal Protocol.** The concept of single-use seals was developed in Peter Todd works on cryptographic commitments and consensus protocols in distributed systems and client-side validation [24, 25, 26, 27]. The protocol for single-use seals is defined in the LNPBP-8 standard [33].

In short, a single-use seal is an abstract mechanism to guarantee a singularity of an event in the future: a mechanism which can be used, for instance, to prevent double-spends. The single-use seal is a commitment to commit to some message that is not known at the time the single-use seal was originally defined. Analogous to the real-world, physical single-use-seals used to secure shipping containers, a single-use-seal primitive is a unique object that can be closed over a message exactly once.

A single-use seal supports two fundamental operations:

(1) $\mathsf{close}(u, m) \to w$: a close of a seal u over a message m, producing a witness w. This fulfills the original commitment, specified in the seal definition.

(2) $\mathsf{verify}(u, w, m) \to \mathbb{B}$: given witness w verify that the seal u was closed over a message m.

A single-use seal protocol is secure if it is impossible for an attacker to cause the verify function to return true for two distinct messages m_1, m_2, when applied to the same seal (although it is acceptable to have multiple witnesses for the same seal / message pair).

RGB contracts use a specific form of the single-use seal protocol, defined by the *layer 1* value (see Table 2) in the contract metadata within the *issue* structure (11). For instance, Bitcoin- and Liquid-based contracts will use TxO-based seals, described in Section 7.1, and Prime-based contracts – seals described in Section 8.

Each RGB operation o_i, including genesis, contains a set of single-use seal definitions $\mathcal{S}_i$, created by that operation (13). These seal definitions must match 1-to-1 with the destructible state $\mathcal{Y}$, defined in the client-side part of the operation c_i, as specified in (14). A pair of the destructible state, defined by an operation, and related single-use seal is called an *assignment*: the state is assigned to the seal. These pairs constitute what is called *owned state* of the *local contract*.

When an operation o_{i+1} references a previous operation o_i output, which defines destructible state, it must provide a *seal close witness* w_{i+1} over the commitment id of the spending operation $\mathsf{Id}(c_i)$, valid for all single-use seals defined and assigned in o_i with the now-destroyed state. This provides the mechanism by which RGB prevents double-spending and guarantees that the destructible state is accessed only by the parties having the rights to access it (*capability-based access*).

5.5. **Set of Operations.** Each contract is defined by a partially ordered set of contract operations $\mathcal{O} \triangleq \{o_i\}$. An element of this set is a tuple, $o_i \triangleq \langle c_i, S_i, u_i \rangle$, consisting of:

- client-side information of contract state change, c_i;
- an unordered map of seal definitions performed by an operation, $S_i : d_j \in \mathcal{Y}_i \to s_j$, where s_j is a seal definition;
- a *seal close witness* information, which values must belong to a set of either unit value, or a specific witness w_i: $u_i \in \{\varnothing, w_i\}$.

The set $\mathcal{O}$ has an initial element, called *genesis* o_0, for which $u_i = \varnothing$. There might be other operations for which $u_i = \varnothing$; these operations are named *state extensions* and must not destroy any owned state (i.e., must not jave any outputs of a destructible type).

All operations in $\mathcal{O}$ are partially ordered using the rule

$$c_i \prec c_j \iff (\exists\ y \in \mathcal{Y}_i : y_{\mathsf{addr}} \in \mathcal{A}_j) \vee (\exists\ x \in \mathcal{X}_i : x_{\mathsf{addr}} \in \mathcal{B}_j) \tag{18}$$

which means that if there exists at least one output of c_i which is used by c_j, or a global state defined in c_i that is read by c_j, then c_i precedes c_j.

If $\mathcal{O}$ is not a directed acyclic graph and the above rule cannot be fulfilled without collisions, the set of operations must be recognized as invalid. This logic is a part of the evaluate procedure (19) and is described in the next section.

5.6. **Evaluate Procedure.** The contract state is evaluated using the following evaluate procedure applied to the set of contract operations:

$$\begin{aligned} \mathsf{evaluate} \triangleq \mathcal{O} \to \mathbb{B} : \\ \forall o_i \in O : \\ (\forall\ a \in \mathcal{A}_i\ \exists!\ d \in \mathcal{D}_i : \mathsf{addr}(d) = a) \\ \wedge\ (\forall\ b \in \mathcal{B}_i\ \exists!\ e \in \mathcal{I}_i : \mathsf{addr}(e) = b) \\ \wedge \begin{cases} w_i = \varnothing : \|\mathcal{A}_i\| \stackrel{?}{=} 0, \\ w_i \neq \varnothing : \mathsf{verify}(\mathcal{S}_i, w_i, \mathsf{Id}(o_i)) \end{cases} \\ \wedge\ \mathsf{exec}(\mathsf{k}, \wp, \langle \mathcal{I}_i, \mathcal{D}_i \rangle, c_i) \\ \Rightarrow \begin{cases} \mathcal{D}_{i+1} \mapsto (\mathcal{D}_i \setminus \mathcal{A}_i) \cup \mathcal{Y}_i, \\ \mathcal{I}_{i+1} \mapsto \mathcal{I}_i \cup \mathcal{Z}_i \end{cases} \\ \Rrightarrow \bot \end{aligned} \tag{19}$$

where the procedures are defined as follows:

- **addr:** returns a memory address for a given memory cell;
- **verify:** performs the verification of a set of single-use seals and a witness according to the LNPBP-8 standard [33] as described in Section 5.4;
- **exec:** executes a verification program and programs verifying fulfillment of individual lock conditions for the spent inputs. The function arguments include codex data k, a set of known AluVM libraries $\wp$, read-only access to the memory $\langle \mathcal{I}, \mathcal{D} \rangle$ and client-side operation data c_i. It succeeds if and only if at the halting of the AluVM machine **CK** register is not set.

The first part of the algorithm (19) checks whether an operation is correct and, in the case of success, the contract state is evolved (middle expression), or the operation must not be applied to the contract state, and further validation must terminate with a failure ($\bot$).

6. Commitments

Commitments are created using strict encoding: a formally-defined binary data serialization algorithm [47]. Specifically:

(1) Numbers are serialized in little-endian format, using a number of bytes required to cover maximal bit dimension of the used numeric type.
(2) Sum types (enums, including primitive types and those with associated data) are prefixed with an 8-bit tag.
(3) Fixed-size arrays are encoded as is, with no prefixing.
(4) Variable-size collections (objects representable as mathematical sets, including sequences, ordered sets, maps) are prefixed with the size of the collection (cardinal number) in little-endian format, using the number of bytes which fully covers the maximal allowed collection dimension; followed by elements of the collection, serialized one after the other.
(5) Totally ordered sets must be serialized in the order of the elements
(6) Partially ordered and unordered sets do not participate in the RGB consensus.
(7) Maps are serialized with elements corresponding to the key and value, composed as a tuple. The keys of the ordered maps must represent a totally

ordered set, and define the order of serialization of key-value tuples.

(8) Product types (tuples) are serialized according to the order of their elements, with no additional prefix of field names.

Data are serialized into hashers, which usually use a prefix (i.e. tagged) to uniquely codify the type of the produced commitment. Collections may also be merklized, in order to allow compact profs of inclusion. The tag is an valid URN [48], which is first hashed with the same hash function, and the result is fed into the new hasher instance twice as its first input, with no length prefix, according to the BIP-340 standard [49].

Version I.0 of the RGB consensus uses the SHA-256 hash function. Future versions may introduce more hashing functions, including zk-friendly, by using feature flags in contract metadata (m_4 field in the contract issue metadata, see Section 5).

The complete specification on the commitment structure is given in the Appendix C.

7. RGB on Bitcoin

When RGB is used on top of Bitcoin [44], or bitcoin-compatible sidechains (such as Liquid, [45]), the following parameters apply:

(1) A contract *issue* must reference the specific bitcoin blockchain & network as a value for its field m_2 (*layer 1* in the contract metadata) using values given in the Table 2.

(2) TxO-based single-use seals must be used (Section 7.1), with Bitcoin transaction outputs serving as seal definitions, and Bitcoin transaction, spending such an output, being a published part of a single-use *seal close witness* (see [33] and Section 7.1 for the definitions of these terms);

(3) Multi-message bundles (Section 7.3) and multiprotocol commitment schemes (Section 7.4) must be used for packing multiple operations under same contract and multiple contracts into a single *seal close witness*.

The TxO single-use seals, multiprotocol commitments and multimessage bundles are vast standards; their complete description would not fit this paper, and, since all these standards are not RGB-specific, their formal specification will be a subject of separate work. Here, we will not be analyzing their properties and will be providing just an overview of their functionality and structure, focusing on how it applies to RGB.

7.1. TxO-based seals.

TxO-based Bitcoin single-use seals are defined in the LNPBP-10 standard [50]. It represents a significant development from the original ideas of Peter Todd [24, 25], bringing support for seal composition into graph structures with their interdependencies, as well as an ability to close a set of seals over a single message, producing a single witness. This was required due to the fact that a single bitcoin transaction output may be used for multiple seal definitions under multiple RGB contracts; and, when it is spent, the spending transaction, becoming a part of the *seal close witness*, needs to contain multiple commitments to multiple operations under many contracts.

7.1.1. *Seal Definitions.* Single-use seals are defined as tuples

$$\mathbb{U} \triangleq \{\mathbb{N}_8^{32}, \varnothing\} \times \mathbb{N}_{32} \times \{\mathbb{N}_8^{40}, \mathbb{N}_8^{32} \times \mathbb{N}_{32}\} \tag{20}$$

$$\mathsf{u} \triangleq \big\langle \{\xi|_{\mathsf{wout}=0}, \langle\zeta, \xi\rangle|_{=1}\}, \{\nu|_{\mathsf{noise}=0}, \langle\zeta', \xi'\rangle|_{\mathsf{fallback}=1}\}\big\rangle \in \mathbb{U} \tag{21}$$

The definition consists of two components:

Primary definition: a Bitcoin transaction outpoint, specified either as an output index $\xi \in \mathbb{N}_{32}$ residing in the same *seal close witness transaction* which closes seals related to the spent inputs (wout variant), or as a complete outpoint, consisting of transaction id and output number $\langle\zeta, \xi\rangle$.

Secondary definition: is either noise data, provided as a means to conceal the seal; or a *fallback seal*, both of which are described below.

The wout case, briefly described above, is required due to the fact that a *seal close witness transaction* id is not known unless the new operation data are fully defined (since it commits to the id of this new operation); hence, if the user wants to reuse the outputs of the *seal close witness transaction* for new seal definitions, the only available information is the transaction output index. Later on, during the verification (19), the transaction id can be automatically resolved as a *seal close witness transaction* id, which was already known at that moment of time.

The noise is provided for privacy reasons: since the *authentication token* α is a hash of the seal definition, and the total number of all UTXOs existing at the moment the operation was created is bound and can be easily enumerated, the noise hides the information about the specific UTXO owned by a user.

When an RGB operation (*spending operation*, o_{i+1}) destroys state created by another operation output, the seals, to which that state was *assigned*, must be closed over the commitment identifier $\mathsf{Id}(c_{i+1})$ of the spending operation (see Section 5.4 for details). This is achieved by putting a deterministic bitcoin commitment to the message derived from the operation id $\mathsf{Id}(o_{i+1})$, into the transaction which spends the transaction outputs participating as seal definitions (named *seal close witness transaction*). The specific way the commitment is produced is described in the following subsections.

The *fallback seals* provide a way how the state assigned to a seal definition may be preserved in case the user spends the seal-defined UTXO without creating a valid *seal close witness*. However, at the version I.0 the *fallback seals* are reserved for the future use; they may be defined, but any spending with a fallback path will be invalid until the consensus version I.1.

7.1.2. *Seal Close Witness. Seal close witness* w_i, defined in 13, for TxO seals consists of two separate components:

***Published seal close witness*:** represented by a *seal close witness transaction*;

***Client-side seal close witness*:** also named *anchor*, includes:

MMB proof: the proof that each witness transaction input is used only in a single RGB operation (see Section 7.3 below);

Contract id and MPC proof: the inclusion proof of an MMB commitment under the

given contract id into the deterministic bitcoin commitment (see Section 7.4 below);

DBC proof: the proof that the witness transaction includes a valid deterministic bitcoin commitment (see Section 7.2 below). This proof is provided only for tapret commitments.

Fallback proof: a constant 1-byte zero value, reserved for future proofs for *fallback seals* use.

7.2. **Deterministic Bitcoin Commitments.** The deterministic bitcoin commitment protocol, defined in the LNPBP-6 standard [51], provides the way to put a provably singular commitment to some external message into a Bitcoin transaction.

(1) The first output of the bitcoin transaction, which is either OP_RETURN or P2TR, must commit to MPC, as defined in Section 7.4 below.
(2) If such an output is an OP_RETURN output, it must contain *opret* type of commitment.
(3) If such an output is a P2TR output, it must contain *tapret* type of commitment.

7.2.1. *Opret commitments.* Opret commitments are created using the following scriptPubkey construction:

(22) `OP_RETURN OP_PUSH32 < MPC_COMMITMENT >`

where `OP_RETURN` is a byte with value $6a_{16}$, `OP_PUSH32` is a byte with value 20_{16}, `<MPC_COMMITMENT>` is a 32-byte little endian binary data of the MPC commitment (see Section 7.4).

7.2.2. *Tapret commitments.* The procedure of tapret commitment is more complex than the one for opret commitment; however, it has several advantages: first, it does not increase the transaction size; second, it does not expose the fact that the Bitcoin transaction contains any associated additional data used outside the Bitcoin protocol, which may be important for privacy reasons, as well as to avoid filtering in Bitcoin transaction relays [52].

The complete commitment protocol is specified in the LNPBP-12 standard [53]. Here, we provide a brief version of its formalism.

First, a tapscript leaf [54], containing the commitment, must be constructed. It consists of 29 bytes with value equal to 61_{64} (`OP_NOP` instruction, representing a no-operation in Bitcoin script), followed by a single `OP_RETURN` byte with value $6a_{16}$, `OP_PUSH32` is a byte with value 20_{16}, 32 bytes of the serialized commitment value (in little-endian order), and a *nonce* byte, which meaning will be explained later. The total length of the tapscript leaf, constructed in such a way, must be equal to 64 bytes.

Next, the constructed tapscript should be embedded in the Taproot script tree, used to construct scriptPubkey of the transaction output [49]. The embedding mechanism depends on the structure of the bitcoin wallet descriptor [55], however, only three options are possible for all descriptor types:

(1) when no script tree is present, the constructed tapleaf script becomes the only leaf in the script tree;
(2) when a script tree contains a single leaf, it is shifted one level below, and is put next to the constructed tapleaf script, containing the commitment;
(3) when a script tree contains multiple leaves, they all shifted one level below, such that the root of the tree becomes a sibling to the constructed tapleaf script.

The resulting tree is used to construct scriptPubkey according to the procedure described in [49].

The tapret DBC proof, included in the *anchor* structure (see Section 7.1.2) consists of the internal public key, used by the output [49], *nonce* value, and an optional information about the second node. This information proves that the partner of the first level of Taproot script tree does not contain an alternative `OP_RETURN` commitment script. In case (1) this information is skipped (since there are no other leaves than the commitment); and in cases (2) and (3) the structure of the information depends on the lexicographic ordering of the tapnode hashes [49] for both of the nodes on the first level. If the leaf tapscript with the commitment precedes the other node, only the tapnode hash of the other node is provided; otherwise, in case (2) the whole leaf script of the other node must be given, and in case (3) two tapnode hashes of the nodes below the sibling node.

The *nonce* value can be used to produce a shorter version of the DBC proof: by trying different *nonce* values one may have a good chance of finding the one which makes tapnode hash of the tapret commitment to precede the other node, shortening the proof to just 32 bytes instead of 64 or more bytes (if the other node contains longer taproot script).

7.3. **Multi-message Bundles.** Multi-message bundles allow multiple independent operations to be packed as a single commitment under a single RGB contract. This is useful in multiparty bitcoin transactions, such as payjoin constructs [56], or multipeer state channels [57].

MMB security achieved by creating an ordered map $\mathcal{T}$ of the bitcoin *seal close witness transaction* input numbers i to the corresponding operation ids $\{\mathsf{Id}(o)\}$ under the contract:

$$\mathcal{T} \triangleq \{i \in \mathbb{N}_{32} \to \mathsf{Id}(o)\}^{[0,2^{32})} \tag{23}$$

This map serves as a proof for MMB commitment and is included as a part of the *client-side seal close witness* (see Section 7.1.2 above).

The map is encoded into the hash function used to produce the commitment value $\beta \in \mathbb{N}_{256}$, which is used later in multiprotocol commitments (see Section 7.4 below).

The complete specification for multimessage bundle commitment scheme can be found in LNPBP-11 standard [58].

7.4. **Multiprotocol Commitments.** Since a single Bitcoin transaction output may be assigned RGB contract state under multiple seal definitions and under multiple RGB contracts, a single Bitcoin transaction spending that output will serve as *seal close witness* to multiple seals. Thus, a special protocol is required to pack the individual operation ids into a single deterministic bitcoin commitment. For this purpose we use a multiprotocol commitment cryptographic scheme, defined in the LNPBP-4 standard [59].

The standard defines the way to commit to multiple independent messages with a single digest such that the fact of each particular commitment and a protocols[4]under which the commitments were made can be proven without exposing the information about other operations and contracts.

Multiple $\beta \in \mathbb{N}_{256}$ hashes of the multimessage bundle under RGB contracts are put into a map

$$\mathcal{M} : \{\mathsf{C}_{\mathsf{Id}} \to \beta\}^{[0,2^{24})} \tag{24}$$

identified with the corresponding contract ids $\{\mathsf{C}_{\mathsf{Id}}\}$.

To construct the multi-protocol commitment an array of 2^s 256-bit natural numbers ($\mathbb{N}_{256}$) is allocated, a random *cofactor* κ value is picked from $\mathbb{N}_{16}$, and each individual β value is put to position ρ in the array defined by

$$\rho = \mathsf{C}_{\mathsf{Id}} \mod \max(2^s - \kappa, 1) \tag{25}$$

In case of a collision, a different κ is picked, or/and the size s is increased, and the operation repeats. The remaining array slots are filled with deterministically generated entropy values, as described in [59].

The resulting array is merklized, and a merkle path to the β commitment, together with the contract id and merklization parameters, is used as a proof of inclusion and singularity of the commitment (see the description of the *anchor* structure in Section 7.1.2 above).

8. RGB on Prime

Prime is a *layer 1* designed specifically to match the requirements of the client-side validation [46]. When combined with zk-STARKs, it may achieve scalability of $O(1)$, independent from a number of transactions and participants [46] — a result which cannot be made possible with traditional blockchains. The use of Prime also allows us to avoid all the complexities associated with the Bitcoin TxO seals construct (Section 7.1).

The complete formal specification of RGB contracts on Prime (RGB′) is outside the scope of this paper and will be part of future work. For now, we will briefly define the main differences from RGB contracts on Bitcoin (RGB₿).

The seal definition for RGB′ is a randomly picked element of $\mathbb{F}_q$. The published part of the *seal close witness* consists of a Prime header [46], and a merkle path proving the inclusion of the RGB operation id into the header serves as the *client-side seal close witness* part. RGB′ does not require the use of multiprotocol commitments or multimessage bundles, which were unavoidable in RGB₿.

9. Reference Implementation

The details on the reference implementation in Rust language are provided in the Appendix B, which are also specified in the RGB-3 standard [60].

10. Acknowledgments

The whole work was inspired by the earlier ideas of Nick Szabo on smart contracts [1], Peter Todd on the client-side validation [23] and single-use seal [24, 25] protocols. Giacomo Zucco was the first to analyze its possible applications and implications for Bitcoin blockchain and Lightning Network [29]. Adam Borko had provided ideas and critics on the zk-STARK compatibility of the protocol and on its operations with Prime, as well as a *fallback seal* mechanism. Olga Ukolova gave a lot of feedback during the protocol design and implementation phase, as well as many years of devoted work to build and educate the community around the project.

Early testers and adopters of the RGB consensus I.0 helped to share and finalize the protocol and provided valuable feedback. Among them, I would like to highlight Bitlight Labs and Armando Dutra.

A great thanks for all the multiple hours of conversations and disputes he had with the thinkers, mathematicians, cryptographers and computer scientists interested in the topic of decentralized systems and cypherpunk; people, who were inspiring the author to continue the work in the most complex times. Among them to be named (in alphabetical order): Alex Kravets, Alexandre Poltorak, Alexis Roussel, Alexis Sellier, Amir Taaki, Andriy Khavryuchenko, Christian Decker, Eric Young, Hunter Beast, Jörg ("Yojoe"), Keith Travin, Konstantin Lomashuk, Rene Pickhardt, Vitaly Bulychov.

This project was supported by donations to LNP/BP Labs from Fulgur Ventures, Hojo Foundation, Pandora Prime SA, Bitlight Labs, DIBA Inc, iFinex Inc and many individual contributors. The author is grateful to investors into Pandora Prime SA, as well as Pandora and Ultraviolet projects investors. which had allowed the research to continue and complete when no other funding sources were available.

The author is grateful to his wife, Orlovska Yassia, and family, who were very supportive during all six years of the work on the project.

References

[1] N. Szabo, "Smart contracts," 1994. [Online]. Available: https://nakamotoinstitute.org/library/smart-contracts/
[2] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Friedenbach, A. Miller, A. Poelstra, J. Timon, and P. Wuille, "Enabling Blockchain Innovations with Pegged Sidechains," 2014. [Online]. Available: https://blockstream.com/sidechains.pdf
[3] S. Lerner, "Drivechains, sidechains and hybrid 2-way peg designs," 2016. [Online]. Available: https://docs.rsk.co/Drivechains_Sidechains_and_Hybrid_2-way_peg_Designs_R9.pdf
[4] X. Chen and K. Zhang, "Extending blockchain functionality with statechain," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, 2017, pp. 240–247.
[5] J. Poon and T. Dryja, "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments," 2016. [Online]. Available: https://lightning.network/lightning-network-paper.pdf
[6] C. Decker, R. Russel, and O. Osuntokun, "eltoo: A Simple Layer2 Protocol for Bitcoin," 2016. [Online]. Available: https://blockstream.com/eltoo.pdf
[7] J. Poon and V. Buterin, "Plasma: Scalable Autonomous Smart Contracts," 2017. [Online]. Available: https://plasma.io/plasma.pdf
[8] "Optimistic rollups," 2025. [Online]. Available: https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/
[9] "Validium," 2025. [Online]. Available: https://ethereum.org/en/developers/docs/scaling/validium/
[10] "Zero-knowledge rollups," 2025. [Online]. Available: https://ethereum.org/en/developers/docs/scaling/zk-rollups/
[11] L. Goldberg, S. Papini, and M. Riabzev, "Cairo – a Turing-complete STARK-friendly CPU architecture," 2021. [Online]. Available: https://eprint.iacr.org/2021/1063.pdf
[12] J. Bonneau, I. Meckler, V. Rao, and E. Shapiro, "Mina: Decentralized Cryptocurrency at Scale," 2020. [Online]. Available: https://minaprotocol.com/wp-content/uploads/technicalWhitepaper.pdf
[13] P. Todd, "Drivechains: A Detailed Analysis," 2023. [Online]. Available: https://petertodd.org/2023/drivechains

[4]In the case of RGB, the "protocols" are respective contract ids C_{Id}; however, multi-protocol commitments allow combining RGB contracts with other unrelated protocols

[14] S. Dziembowski, G. Fabiański, S. Faust, and S. Riahi, "Lower bounds for off-chain protocols: Exploring the limits of plasma," Cryptology ePrint Archive, Paper 2020/175, 2020. [Online]. Available: https://eprint.iacr.org/2020/175

[15] L. Donno, "Optimistic and validity rollups: Analysis and comparison between optimism and starknet," 2022. [Online]. Available: https://arxiv.org/abs/2210.16610

[16] R. Pickhardt and M. Nowostawski, "Imbalance measure and proactive channel rebalancing algorithm for the lightning network," 2019. [Online]. Available: https://arxiv.org/abs/1912.09555

[17] F. Waugh and R. Holz, "An empirical study of availability and reliability properties of the bitcoin lightning network," 2020. [Online]. Available: https://arxiv.org/abs/2006.14358

[18] R. Pickhardt, S. Tikhomirov, A. Biryukov, and M. Nowostawski, "Security and privacy of lightning network payments with uncertain channel balances," 2021. [Online]. Available: https://arxiv.org/abs/2103.08576

[19] P. Zabka, K.-T. Förster, C. Decker, and S. Schmid, "A centrality analysis of the lightning network," *Telecommunications Policy*, vol. 48, no. 2, p. 102696, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0308596123002070

[20] P. Zabka, K.-T. Foerster, S. Schmid, and C. Decker, "A centrality analysis of the lightning network," 2022. [Online]. Available: https://arxiv.org/abs/2201.07746

[21] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014-2025. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[22] B. David, P. Gaži, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol," Cryptology ePrint Archive, Paper 2017/573, 2017. [Online]. Available: https://eprint.iacr.org/2017/573

[23] P. Todd, "Progress on scaling via client-side validation," 2016. [Online]. Available: https://milan2016.scalingbitcoin.org/files/presentations/D2%20-%20A%20-%20Peter%20Todd.pdf

[24] ——, "Building blocks of the state machine approach to consensus," 2016. [Online]. Available: https://petertodd.org/2016/state-machine-consensus-building-blocks

[25] ——, "Preventing consensus fraud with commitments and single-use-seals," 2016. [Online]. Available: https://petertodd.org/2016/commitments-and-single-use-seals

[26] ——, "Closed seal sets and truth lists for better privacy and censorship resistance," 2016. [Online]. Available: https://petertodd.org/2016/closed-seal-sets-and-truth-lists-for-privacy

[27] ——, "Scalable semi-trustless asset transfer via single-use-seals and proof-of-publication," 2017. [Online]. Available: https://petertodd.org/2017/scalable-single-use-seal-asset-transfer

[28] M. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. Jones, and P. Wadler, *The Extended UTXO Model*, 08 2020, pp. 525–539.

[29] G. Zucco, "Giacomo Zucco and RGB tokens built on bitcoin," 2016. [Online]. Available: https://bitcoinmagazine.com/culture/video-interview-giacomo-zucco-rgb-tokens-built-bitcoin

[30] M. Orlovsky, "AluVM: a functional RISC virtual machine," 2022. [Online]. Available: https://docs.aluvm.org

[31] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," Cryptology ePrint Archive, Paper 2018/046, 2018. [Online]. Available: https://eprint.iacr.org/2018/046

[32] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed systems*. Pearson Education, 2011.

[33] P. Todd, "Single-use seals," 2025. [Online]. Available: https://github.com/LNP-BP/LNPBPs/blob/master/lnpbp-0008.md

[34] K. Rosen, *Discrete Mathematics and Its Applications*. McGraw-Hill Education, 2006. [Online]. Available: https://books.google.ch/books?id=ZYZfQgAACAAJ

[35] M. Orlovsky, "Versioning of RGB releases," 2025. [Online]. Available: https://github.com/RGB-WG/RFC/blob/master/RGB-0006.md

[36] J. von Neumann, "First draft of a report on the edvac," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[37] T. Noergaard, *Embedded Systems Architecture*, ser. Embedded Technology. Burlington: Newnes, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780750677929500005

[38] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," 1999. [Online]. Available: https://www.cs.utexas.edu/~shmat/courses/cs380s/cowan.pdf

[39] N. Imtiaz and L. Williams, "Memory error detection in security testing," 2021. [Online]. Available: https://arxiv.org/abs/2104.04385

[40] N. S. Agency, "Software Memory Safety, Cybersecurity Information Sheet," 2023. [Online]. Available: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/1/CSI_SOFTWARE_MEMORY_SAFETY.PDF

[41] M. Orlovsky, "Rust crate implementing zk-AluVM," 2025. [Online]. Available: https://docs.rs/zk-aluvm

[42] ——, "Rust crate implementing AluVM," 2025. [Online]. Available: https://docs.rs/aluvm

[43] ——, "RGB-I-0.1-0 ABIs," 2025. [Online]. Available: https://github.com/RGB-WG/RFC/blob/master/RGB-1010.md

[44] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://cdn.nakamotoinstitute.org/docs/bitcoin.pdf

[45] J. Nick, A. Poelstra, and G. Sanders, "Liquid: A bitcoin sidechain," 2020. [Online]. Available: https://blockstream.com/assets/downloads/pdf/liquid-whitepaper.pdf

[46] M. Orlovsky, "Scaling and anonymizing bitcoin at layer 1 with client-side validation," 2023. [Online]. Available: https://github.com/LNP-BP/layer1

[47] ——, "Strict types: Portable & deterministic formalism for algebraic data types," 2023. [Online]. Available: https://docs.strict-types.org

[48] P. Saint-Andre and D. J. C. Klensin, "Uniform Resource Names (URNs)," RFC 8141, Apr. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8141

[49] P. Wuille, J. Nick, and A. Towns, "Taproot: SegWit version 1 spending rules," 2020. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki

[50] M. Orlovsky, P. Todd, G. Zucco, F. Tenga, and A. Borko, "Bitcoin transaction output-based single-use-seals," 2025. [Online]. Available: https://github.com/LNP-BP/LNPBPs/blob/master/lnpbp-0010.md

[51] M. Orlovsky, "Deterministic bitcoin commitments," 2025. [Online]. Available: https://github.com/LNP-BP/LNPBPs/blob/master/lnpbp-0006.md

[52] L. Wright, "Bitcoin's cold war: nearly 3,000 nodes at risk as policy tensions escalate ahead of next Bitcoin Core release," 2025. [Online]. Available: https://cryptoslate.com/bitcoins-cold-war-nearly-3000-nodes-at-risk-as-policy-tensions-escalate-ahead-o?utm_source=chatgpt.com

[53] M. Orlovsky and P. Todd, "TapRet: Taproot script tree-based OP_RETURN deterministic bitcoin commitments," 2022. [Online]. Available: https://github.com/LNP-BP/LNPBPs/blob/master/lnpbp-0012.md

[54] P. Wuille, J. Nick, and A. Towns, "Validation of Taproot Scripts," 2020. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0342.mediawiki

[55] P. Wuille and A. Chow, "Output Script Descriptors General Operation," 2021. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0380.mediawiki

[56] C. Belcher, R. Masutti, and D. De Rosa, "PayJoin," 2021. [Online]. Available: https://en.bitcoin.it/wiki/PayJoin

[57] M. Orlovsky, "Nucleus: capital-efficient multipeer lightning payment channels," 2023.

[58] ——, "Anchoring multiple deterministic bitcoin commitments in the same transaction output," 2024. [Online]. Available: https://github.com/LNP-BP/LNPBPs/blob/master/lnpbp-0011.md

[59] ——, "Multi-message commitment scheme with zero-knowledge provable unique properties," 2025. [Online]. Available: https://github.com/LNP-BP/LNPBPs/blob/master/lnpbp-0004.md

[60] "List of RGB repositories and libraries," 2025. [Online]. Available: https://github.com/RGB-WG/RFC/blob/master/RGB-0003.md

Appendix A. RGB Instruction Set Architecture

Table 5. AluVM Control Flow Instruction Set Architecture

Op. code	Instr. byte	Args	Src.	Dst.	**CK** change	**CS** call stack	**ΔCA**	Description
NOP	0	-	-	-	-		0	Not an operation
NOCO	1	-	**CO**	**CO**	-		2000	Inverts **CO**
CHCO	2	-	**CO**	-	-		2000	Terminate if **CO** $\stackrel{?}{=}$ true
CHCK	3	-	**CK**	-	-		2000	Terminate if **CK** is in failed state
FAIL	4	-	-	-	failed		2000	Set **CK** to failed state, terminates if **CH** is set
RSET	5	-	**CK**	**CO**, **CK**	-		2000	Reset **CK**, set **CO** to the pre-reset **CK** value
JMP	6	p [1]	-	-	may fail		10000	Unconditional jump to location
JINE	7	p	**CO**	-	may fail		20000	Jump to location if **CO** $\stackrel{?}{=}$ true
JIFAIL	8	p	**CK**	-	may fail		20000	Jump to location if **CK** is in failed state
SH	9	x [2]	-	-	may fail		10000	Relative jump
SHNE	10	x	**CO**	-	may fail		20000	Relative jump if **CO** $\stackrel{?}{=}$ true
SHFAIL	11	x	**CK**	-	may fail		20000	Relative jump if **CK** is in failed state
EXEC	12	$\mathsf{h} \in \mathbb{E}$	-	-	may fail		548000	Jump outside the current library to an entry point
FN	13	p	-	-	may fail	pushes	30000	Call inside the current library
CALL	14	$\mathsf{h} \in \mathbb{E}$	-	-	may fail	pushes	548000	Call an entry point outside the current library
RET	15	-	-	-	-	pops	20000	Return or finish the program
STOP	16	-	-	-	-		0	Unconditionally stop the program execution

[1] absolute position (a $\mathbb{N}_{16}$ natural number) in the library
[2] relative offset (a $\mathbb{Z}_8$ integer) from the current position in the library

TABLE 6. **GFA256** Instruction Set Architecture Extension for AluVM

Op. code	Instr. byte	Extension half-byte	Src. Args.	Dst. Args.[1]	**CK** change	**ΔCA**	Description
TEST	64	0000_2	**E**	(**CO**)		256000	Tests if the register contains a value. Set **CO** to the result of the test.
CLR	64	0001_2		**E**		256000	Clear the destination register value.
PUTD	64	0010_2	$f \in \mathbb{F}_q$	**E**		768000	Put a given value to the destination register.
PUTZ	64	0011_2		**E**		256000	Put zero value to the destination register.
PUTV	64	0100_2|const		**E**		264000	Put predefined constant value to the destination register. The pre-defined constant is a $\mathbb{N}_2$ number, which values represent 1, 2^{64}, 2^{128} and q (the order of the finite field taken from **FQ**).
FITS	64	1000_2|bits	**E**	(**CO**)	may fail	264000	Test whether a value in the source register fits in the provided number of bits. Set **CO** register to the results of the test. The bits value is a $\mathbb{N}_3$ number representing the maximal number of lowest bits which may be set. If the source register does not contain a value, set both **CO** and **CK** to the failed state; otherwise leaves the value in **CK** unchanged.
MOV	65	-	**E**	**E**, (**CO**)		512000	Move (copy) value from the source to the destination register, overwriting the previous value in the destination. If the source has no value, the destination value is also cleared. Operation leaves the value in the source register unaffected, and does not change the values in **CO** and **CK** registers.
EQ	66	-	**E**, **E**	(**CO**)	may fail	512000	Set **CO** to the result of register equivalence test. If both registers do not contain a value, set **CK** to the failed state.
NEG	67	-	**E**	**E**	may fail	512000	Take the value from the source register, negate using finite-field modulo arithmetics, and put the result into the destination register. If the source register has no value **CK** is set to the failed state.
ADD	68	-	**E**, **E**	**E**	may fail	512000	Add values of two registers using finite-field arithmetics and put the result into the first of the registers. Do not detect overflows. If either of the source register does not contain a value, sets **CK** fo the failed state.
MUL	69	-	**E**, **E**	**E**	may fail	768000	Multiply values of two registers using finite-field arithmetics and put the result into the first of the registers. Do not detect overflows. If either of the source register does not contain a value, sets **CK** fo the failed state.

E represents any of **E1-E8** and **EA-EH** general-purpose registers.

The value of the explicit arguments are encoded after the instruction byte and half-bit, as a bit stream using the little-endian format. **E** registers are represented as $\mathbb{N}_4$ numbers, starting from **E1**.

[1] Implicit args are given in parenthesis

TABLE 7. **USONIC** Instruction Set Architecture Extension for AluVM

Op. code	Instr. byte	Src.	Dst.	Description
CKNXIRO	128	$\mathcal{A}$, **UI1**	**CO**	Set **CO** to the result of check **UI1** $< \|\mathcal{A}\|$
CKNXIAO	129	$\mathcal{B}$, **UI2**	**CO**	Set **CO** to the result of check **UI2** $< \|\mathcal{B}\|$
CKNXORO	130	$\mathcal{Y}$, **UI3**	**CO**	Set **CO** to the result of check **UI3** $< \|\mathcal{Y}\|$
CKNXOAO	131	$\mathcal{Z}$, **UI4**	**CO**	Set **CO** to the result of check **UI4** $< \|\mathcal{Z}\|$
LDW	132	Υ		Load operation witness data (not the same as single-use seal witness!) to **EA-ED** registers.
LDIW	133	$\Omega\|_{\mathsf{addr}=\mathsf{a}_{\mathbf{UI1}}}$		Load a destructible *input-specific witness* Ω for the current input number (determined by the value in **UI1**) to **EA-ED** registers. The operation is idempotent and does not update the value of **UI1** register. It also sets **CO** to indicate whether the input matching **UI1** value is present.
LDIL	134	$\mathsf{b}_{\Theta}\|_{\mathsf{addr}=\mathsf{a}_{\mathbf{UI1}}}$		Load a destructible *input-specific auxiliary locking conditions* Θ for the current input number (determined by the value to the **UI1**) to **EA-ED** registers. The operation is idempotent and does not update the value of **UI1** register. It also sets **CO** to indicate whether the input matching **UI1** value is present.
LDIT	135	$\alpha_{\mathbf{UI1}}$	**CO**	Load a destructible input authorization token α for the current input number (determined by the value in **UI1**) to **EA**. The operation also sets **EB** to either 0 (if a custom lock script is not present) or 1, if it is present. The operation is idempotent and does not update the value of **UI1** register. It also sets **CO** to indicate whether the input matching **UI1** value is present.
LDIRO	136	$\mathsf{d}_{\sigma}\|_{\mathsf{addr}=\mathsf{a}_{\mathbf{UI1}}}$	**EA-ED**, **CO**	Load the destructible memory cell referenced by the next operation input to **EA-ED** registers. Set **CO** to indicate whether the next element was present.
LDIAO	137	$\mathsf{e}_{\varsigma}\|_{\mathsf{addr}=\mathsf{b}_{\mathbf{UI2}}}$	**EA-ED**, **CO**	Load the immutable memory cell referenced by the next operation input to **EA-ED** registers. Set **CO** to indicate whether the next element was present.
LDORO	138	$\sigma(\mathsf{y}_{\mathbf{UI3}})$	**EA-ED**, **CO**	Load next destructible operation output value to **EA-ED** registers. Set **CO** to indicate whether the next element was present.
LDOAO	139	$\varsigma(\mathsf{z}_{\mathbf{UI4}})$	**EA-ED**, **CO**	Load next immutable operation output value to **EA-ED** registers. Set **CO** to indicate whether the next element was present.
RSTIRO	140	-	**UI1**	Resets iterator over the input destructible memory cells by setting **UI1** to zero.
RSTIAO	141	-	**UI2**	Resets iterator over the input immutable memory cells by setting **UI2** to zero.
RSTORO	142	-	**UI3**	Resets iterator over the output destructible memory cells by setting **UI3** to zero.
RSTOAO	143	-	**UI4**	Resets iterator over the output immutable memory cells by setting **UI4** to zero.

All **USONIC** instructions do not take arguments and have zero complexity. None of them changes the value in **CK**.

Appendix B. RGB Consensus Reference Implementation

In the lists below, the top level contains a reference to a repository, and the layers underneath describe specific Rust crates with library and binary targets. If the crates are not listed, then the repository contains a single crate named after it.

B.1. **Client-side validation library.** `https://github.com/LNP-BP/client_side_validation`
Provides core cryptographic primitives for client-side validation (not specific to RGB or Bitcoin).

- `commit_verify`: implementation of tagged hashing, merklization, commitment creation workflows, multiprotocol commitments and multimessage bundles.
- `commit_encoding_derive`: derivation macros for applying commitment workflows to data structures.
- `single_use_seals`: single-use seal APIs.
- `client_side_validation`: umbrella library combining the above.

B.2. **Strict Encoding.** `https://github.com/strict-types/strict_encoding`
Consensus data serialization procedures.

- `strict_encoding`: serialization implementation.
- `strict_encoding_derive`: derivation procedural macros.

B.3. **zk-AluVM.** `https://github.com/AluVM/aluvm` and `https://github.com/AluVM/zk-aluvm`
Virtual machine for RGB (see Section 4.1)

- `aluvm`: framework for building virtual machines.
- `zk-aluvm`: zk-compatible RISC virtual machine.

B.4. **UltraSONIC.** `https://github.com/AluVM/ultrasonic`
Consensus-level part of the polynomial computer based on zk-AluVM (see Section 4).

B.5. **BP Core Library.** `https://github.com/BP-WG/bp-core`
Implementation of Bitcoin TxO-based single-use seals and commitment schemes.

- `bp-consensus`: Bitcoin blockchain consensus-level primitives,
- `bp-dbc`: deterministic bitcoin commitments (opret, tapret),
- `bp-seals`: bitcoin-specific single-use seals code,
- `bp-core`: umbrella library combining all of the above.

B.6. **RGB Core Library.** `https://github.com/RGB-WG/rgb-core`
RGB I.0 consensus implementation orchestrates the operations of the above libraries.

Appendix C. Commitment Structure

C.1. **Codex Identifier.**

```
commitment CodexId: for Codex, hasher SHA256, tagged urn:ubideco:sonic:codex#2025-05-15
  serialize Codex
    bytes version: len 1, alias ReservedBytes1
    str name: len 0..=FF.h
    ascii developer: alias Identity, first AsciiPrintable, rest AsciiPrintable, len 1..=1000.h
    field timestamp: I64
    bytes features: len 4, alias ReservedBytes4
    field fieldOrder: U256
    struct verificationConfig: CoreConfig
      enum halt: Bool, false 0, true 1
        field some: U64, option, wrapped, tag 1
    struct inputConfig: CoreConfig
      enum halt: Bool, false 0, true 1
        field some: U64, option, wrapped, tag 1
    map verifiers: len 0..=FF.h
      field key: U16
      struct value: LibSite
        bytes libId: len 32, alias LibId
        field offset: U16
```

C.2. **Contract Identifier.**

```
commitment ContractId: for Issue, hasher SHA256, tagged urn:ubideco:sonic:contract#2024-11-16
  serialize version: ReservedBytes1
    const 0: U8
  serialize meta: ContractMeta
    enum testnet: Bool, false 0, true 1
    enum consensus: Consensus, none 0, bitcoin 16, liquid 17, prime 32
    field timestamp: I64
    bytes features: len 6, alias ReservedBytes6
    union name: ContractName
      field unnamed: Unit, tag 0
      ascii named: wrapped, alias TypeName, first AlphaCapsLodash, rest AlphaNumLodash, len 1..=64.h, tag 1
    ascii issuer: alias Identity, first AsciiPrintable, rest AsciiPrintable, len 1..=1000.h
  serialize codexId: commitment CodexId
  serialize genesisId: commitment OpId
```

C.3. **Genesis Operation Identifier.**

```
commitment Opid: for Genesis, hasher SHA256, tagged urn:ubideco:ultrasonic:operation#2024-11-14
  serialize version: ReservedBytes1
    const 0: U8
  serialize codexId: commitment, alias CodexId
  serialize callId: U16
  serialize nonce: Fe256
  serialize opWitness: StateValue
  merklize destructibleIn: [Input ^ 0]
  merklize immutableIn: [CellAddr ^ 0]
  merklize destructibleOut: [StateCell ^ 0..<2^16]
    serialize StateCell
      union data: StateValue
      field auth: U256, alias AuthToken, alias Fe256
      struct lock: option, wrapped CellLock, tag 1
        union aux: StateValue
        struct script: LibSite, option, wrapped, tag 1
          bytes libId: len 32, alias LibId
          field offset: U16
  merklize immutableOut: [StateData ^ 0..<2^16]
    serialize StateData
      union value: StateValue
      bytes raw: len 0..<2^16, option, wrapped RawData, tag 1
```

C.4. **Operation Identifier.**

```
commitment Opid: for Operation, hasher SHA256, tagged urn:ubideco:ultrasonic:operation#2024-11-14
  serialize version: ReservedBytes1
    const 0: U8
  serialize contractId: commitment, alias ContractId
  serialize callId: U16
  serialize nonce: Fe256
  serialize opWitness: StateValue
  merklize destructibleIn: [Input ^ 0..<2^16]
    serialize Input
      struct addr: CellAddr
        bytes opid: len 32, alias Opid
        field pos: U16
      union witness: StateValue
  merklize immutableIn: [CellAddr ^ 0..<2^16]
    serialize CellAddr
      bytes opid: len 32, alias Opid
      field pos: U16
  merklize destructibleOut: [StateCell ^ 0..<2^16]
    serialize StateCell
      union data: StateValue
      field auth: U256, alias AuthToken, alias Fe256
      struct lock: option, wrapped CellLock, tag 1
        union aux: StateValue
        struct script: LibSite, option, wrapped, tag 1
          bytes libId: len 32, alias LibId
          field offset: U16
  merklize immutableOut: [StateData ^ 0..<2^16]
    serialize StateData
      union value: StateValue
      bytes raw: len 0..<2^16, option, wrapped RawData, tag 1
```

Appendix D.

Glossary

anchor: a *client-side seal close witness* for RGB on Bitcoin. 9–11, 19
assignment: a pair of single-use seal definition and a *destructible memory* cell. 8, 19
authentication token: unique identifier α, an element of the finite field $\mathbb{F}_q$, used for capability-based access to write-once memory cell. 4, 9, 19

client-side seal close witness: a part of *seal close witness*, which is not available for public, and must be kept by the parties of the RGB contract on their local machines. The parties exchange these data via P2P interaction. See also *anchor*. 9–11, 19
composed value: a variable length sequence of field elements in $\mathbb{F}_q$, from zero to four. 4

destructible memory: see *read-once memory*. 4, 19

fallback seal: an part of single-use seal definition, which defines a second seal, used if the first seal was not properly closed. 9–11

genesis: The first operation in RGB contract, which becomes part of the contract *issue*. 7, 8

immutable memory: see *write-once memory*. 4, 19
input-specific witness: optional data Ω, provided per operation input, which may be used to satisfy *locking condition* to gain access to destructible memory cells. Unlike *operation-level witness* these data apply to a single input. Not related to *seal close witness*. 4, 7, 19
issue: data structure defining unique RGB contract. 5, 6, 8, 9, 19

layer 1: in context of RGB, a part of the consensus protocol, operating as a public network. May be represented by a blockchain network, or a new type of network, such as *Prime*. 2, 6, 8, 9, 11, 19
local contract: a part of RGB contract operation history $\mathcal{O}$ known on the local node, as a result of partial contract replication. 5, 8
local contract state: a state known on the local node, as a result of partial contract replication. 6, 7, 19
locking condition: data structure ℓ, defining access rights to an *immutable memory* cell. Consist of an optional program entry point ε and auxiliary data Θ. The script must be satisfied with Θ, an operation-wide witness and/or witness in the spending operation input (see *operation-level witness* and *input-specific witness*). 4, 19

operation-level witness: Optional data Υ, provided in the client-side part of an RGB operation, which may be used to satisfy *locking condition* to gain access to destructible memory cells. Unlike *input-specific witness* these data apply to all inputs. Not related to *seal close witness*. 7, 19
opret: a form of deterministic bitcoin commitment used in *seal close witness transaction*, produced using a dedicated `OP_RETURN` output. 10
owned state: RGB *local contract state*, consisting of individual *assignments*. 4, 8

Prime: a *layer 1* designed specifically to match the requirements of the client-side validation, which has a theoretical scalability of $O(1)$, i.e., independent of the number of transactions and participants. 19
published seal close witness: a part of *seal close witness*, which is available for public, i.e., is included in some *layer 1* network data structures. See also *seal close witness transaction*. 9, 19

read-once memory: capability-based memory, an element of which ("cell") can be read only once, by providing an *authentication token* and fulfilling optional *locking condition*. Once accessed, the memory cell is destroyed. 4, 7, 19

seal close witness: w_i proving that the single-use seal was indeed closed over some specific message. Consists of *published seal close witness* and *client-side seal close witness* parts. See also *seal close witness transaction*. 7–11, 19
seal close witness transaction: a *published seal close witness* for RGB on Bitcoin. 9, 10, 19
state extension: a type of RGB operation, which has no inputs referencing destructible state, and no *seal close witness*. 7, 8

tapret: a form of deterministic bitcoin commitment used in *seal close witness transaction*, produced using Taproot script tree. 10

write-once memory: memory, an element of which ("cell") can be created only once, cannot be changed or destroyed, and can be accessed (read) multiple times by any party. 7, 19