Specular: Towards Trust-minimized Blockchain Execution Scalability with EVM-native Fraud Proofs

Zhe Ye*, Ujval Misra* and Dawn Song
University of California, Berkeley

***Abstract*—**An optimistic rollup (ORU) enables refereed delegation of computation from a blockchain (L1) to an untrusted remote system (L2), by allowing state updates posted on-chain to be disputed by any party via an *interactive fraud proof* (IFP) protocol. Existing systems that utilize this technique have demonstrated up to a 20x reduction in transaction fees.

The most popular ORUs today, in active development, strive to extend existing Ethereum client software to support IFP construction, aiming to reuse prior L1 engineering efforts and replicate Ethereum Virtual Machine (EVM) semantics at L2. Unfortunately, to do so they tightly couple their on-chain IFP verifier with a *specific client program binary*—oblivious to its higher-level semantics. We argue that this approach (1) precludes the trust-minimized, permissionless participation of multiple Ethereum client programs, magnifying monoculture failure risk; (2) leads to an unnecessarily large and complex trusted computing base that is difficult to independently audit; and, (3) suffers from a frequently-triggered, yet opaque upgrade process—both further increasing auditing overhead, and complicating on-chain access control.

In this work, we aim to build a secure, trust-minimized ORU that addresses these problems, while preserving scalability and dispute resolution efficiency. To do so, we design an IFP system *native* to the EVM, that enforces Ethereum's specified semantics precisely at the level of a single EVM instruction.

We present *Specular*, an ORU which leverages an off-the-shelf Ethereum client—modified *minimally* to support IFP construction—demonstrating the practicality of our approach.

1. Introduction

Public blockchains, such as Ethereum [1], have struggled to scale with growing demand, resulting in exorbitant transaction fees during congestion. A recent line of work [2] has explored a promising class of solutions that aims to mitigate this problem by offloading transaction execution to a more powerful off-chain system (L2) run by untrusted parties (*validators*), while the underlying trusted blockchain (L1) efficiently confirms the validity of the result.

One particular approach in this class, the *optimistic rollup* (ORU), operates in the refereed games model [3], requiring an L2 validator to post and attest to a cryptographic commitment on L1 asserting the validity of the L2 virtual machine (L2VM) state, resulting from the execution of off-chain transactions. This commitment is optimistically accepted on L1, but can be disputed on-chain by challengers for a period of time before it is confirmed and considered final. The dispute resolution protocol consists of an *interactive fraud proof* (IFP), which requires the challenger to participate in an interactive game with the defending validator and submit a proof of an invalid state transition (*one-step proof*). The commitment is eventually rejected only if the verifier accepts this proof. This mechanism guarantees the validity of the rollup state, assuming a single honest validator exists.

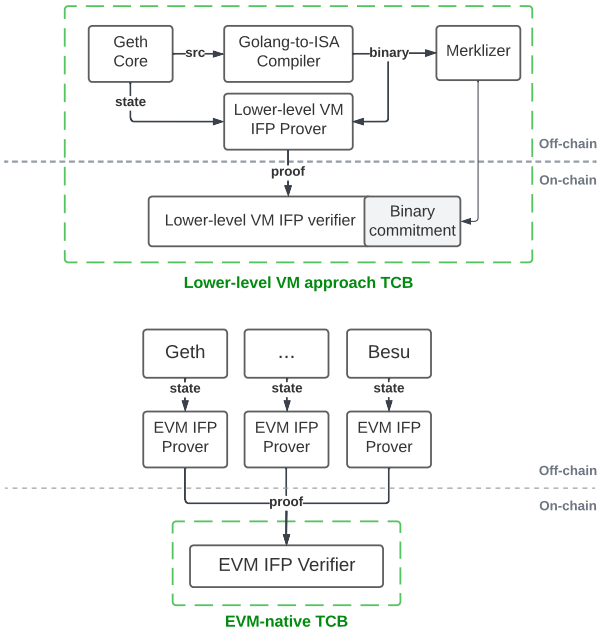


Figure 1: TCB comparison. The top diagram illustrates the complexity and size of the TCB in the lower-level VM approach. The bottom diagram illustrates the simplicity and minimal size of the TCB in the EVM-native approach.

Recent work by Arbitrum and Optimism, the most popular ORUs deployed today, utilizes an architecture [4], [5] that reuses existing Ethereum client infrastructure, in an attempt to replicate EVM semantics at L2, and benefit from prior engineering and security auditing efforts contributed at L1. Specifically, they adopt and modify an existing Ethereum client, called Go-Ethereum (Geth) [6]. To support IFPs, the Geth EVM Golang source is compiled to the instruction set architecture (ISA) of a lower-level target VM. The resulting

* *Equal contribution.*

binary is then committed to on-chain as a Merkle root, allowing the verifier to verify membership of an instruction, and simulate its execution.

Crucially, this approach does not directly enforce EVM semantics; in fact, the verifier is entirely unaware of the EVM's existence. Instead, it enforces the execution of a *specific client binary* that must be trusted to have implemented the EVM. This has three key disadvantages: it (1) precludes the trust-minimized, permissionless participation of multiple Ethereum client programs, hindering client diversity; (2) leads to an unnecessarily large and complex trusted computing base (TCB) that is difficult to independently audit and impractical to formally verify; and, (3) suffers from a frequently triggered upgrade process, magnifying security audit overhead and complicating contract access control.

First, by binding the verifier to a specific Ethereum client program, other L1 client programs are precluded from participating at L2, fundamentally prohibiting *N-version programming* [7]. L2 networks that take this approach are hence prone to monoculture failures. At L1, the Ethereum network has demonstrated resilience despite occasional mass client failures caused by software bugs [8], [9], [10], owing to the diversity of participating clients[1]. Existing L2 approaches have no comparable safeguards. Consequently, invalid state transitions induced by software bugs can slip by undisputed—ultimately resulting in loss of user funds.

Second, because the L1 verifier contract is tasked with verifying the execution of the client at the target ISA instruction-level (as opposed to higher-level EVM semantics), the TCB includes the source code of the entire client, Golang-to-ISA compiler and binary commitment generator (as illustrated in Figure 1). This leads to a larger attack surface and commensurately increases auditing overheads. Formal verification against an EVM specification [11] is also infeasible in this regime, given the unbounded and concurrent nature of these programs.

Third, auditing overheads are compounded by the high frequency in upgrades resulting from a large TCB. Client programs, for example, are upgraded more frequently than the underlying EVM specification itself [6], [12]. Moreover, *any* off-chain component upgrade requires an on-chain upgrade of the binary commitment in-tandem, putting contract governance on the critical path. Such upgrades not only require developer trust, but are also often slow, controversial and disruptive to the system [13]. For this reason, ORU project developers have expressed a desire to forfeit L1 contract upgrade control in the long-term [14]. However, this remains risky and unrealistic, given all off-chain component upgrades—including vulnerability patches—must be reflected on-chain in a new binary commitment.

Requirements. An ORU should therefore fulfill four high-level design goals. First, the ORU architecture should facilitate—not hinder—client diversity, taking advantage of prior engineering efforts in the Ethereum ecosystem. Validators should not require permission to use their choice of L2 client program. Second, the TCB must be sufficiently small and simple, to enable effective security audits and ideally, formal verification against a formal specification. Third, TCB upgrades should be infrequent, while client program upgrades which do not affect semantics should not be hindered by L1 governance. Last, it must be efficient during both normal execution and dispute resolution. We focus our efforts on scaling Ethereum specifically, since it is the largest public programmable blockchain and offers client diversity as a reusable resource; however, we note that this work generalizes to other blockchains (trivially in the case of EVM-based chains).

1. In the cited case, the deciding factor was software *version* diversity.

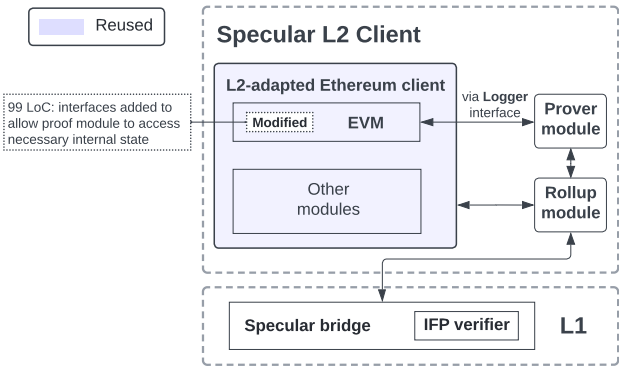


Figure 2: Specular L2 client overview. The L2 client is composed of an L2-adapted Ethereum client (e.g. Geth), modifying the EVM to support proof generation; a proof module that generates one-step proofs; and a rollup module that is responsible for sequencing and validation.

EVM-native IFP. We therefore argue that the verification of semantics beyond those defined in the EVM specification is counterproductive, due to the disadvantages outlined. We instead propose an *EVM-native* interactive fraud proof approach, which directly verifies EVM semantics. That is, we propose to generate an efficiently verifiable one-step proof *directly* over the state transition resulting from the execution of a *single EVM instruction*. As such, it is feasible to construct the proof with any existing Ethereum client program. This differs from other systems, which target alternative ISAs (see Table 1 for a full comparison). We claim that unlike those efforts, an approach under the proposed paradigm can address all of the requirements outlined.

An EVM-native IFP system reduces barriers to L2 client diversity: multiple Ethereum clients [6], [19], [20] already implement the EVM specification, and by design can be made compatible with such a system with relatively little effort, as we demonstrate in this work. Because the proof system completely preserves EVM semantics at the instruction-level, *any* Ethereum client can validate state commitments and participate in dispute resolution—as opposed to ongoing efforts [5], [4], which suffer from client lock-in.

Moreover, this design reduces the scope of the TCB to *only* the EVM IFP verifier on L1. The TCB is therefore minimal in size, practically improving auditability. Additionally, this results in fewer upgrades; specifically, only upgrades to the EVM specification can trigger a TCB upgrade. Client

System	Properties			
	Fraud proof interactivity	Efficient dispute resolution	One-step proof target ISA	EVM reuse
Optimism [15]	Non-interactive	✗	N/A	EVM-compatible
Arbitrum [16]	Interactive	✓	AVM [16]	EVM-compatible
Optimism Cannon† [4]	Interactive	Pending impl.	MIPS [17]	EVM-equivalent
Arbitrum Nitro [5]	Interactive	✓	WebAssembly [18]	EVM-equivalent
Specular	Interactive	✓	EVM [1]	EVM-native

TABLE 1: System comparison. Efforts in the Optimism and Arbitrum communities on Cannon and Nitro respectively are ongoing (as of writing). † To our knowledge, Cannon does not yet have a fully specified and working challenge protocol.

program upgrades can be made independently of the TCB, reducing governance overhead and making forfeiture of L1 contract ownership more plausible.

Contributions. In this work, we make both conceptual and technical contributions in designing and developing a new optimistic rollup. In Section 3, we propose the use of an EVM-native IFP and motivate its necessity in the context of prior work. In Section 4 and Section 5, we provide the first concrete scheme and implementation for an EVM-native IFP. The key challenge in our approach is to support one-step proof generation directly over all EVM instructions and inter-transaction operations, and doing so without significantly modifying client internals. The conventional wisdom in the blockchain industry has held the sentiment that to do so would pose an overwhelming challenge, due to the general complexity of the EVM instruction set [21], [22]. However, we show its feasibility by formulating the one-step proof for each instruction's execution as a modular composition of a few simple subproofs, each of which proves the validity of a state transition in the EVM stack, memory, persistent storage or other auxiliary data structures.

We introduce *Specular*, an end-to-end implementation of an ORU with an EVM-native IFP system, demonstrating the practicality of our approach.

In summary, we make the following key contributions:

- We propose using an *EVM-native* IFP approach that verifies only EVM semantics and nothing more, to enable a minimal TCB and mitigating barriers to trust-minimized, permissionless client diversity.
- We introduce a concrete EVM-native, efficiently verifiable one-step proof construction that supports all requisite opcodes and inter-transaction operations.
- We adapt the EVM implementation of the most popular Ethereum client, Geth [6], to the IFP setting. We modify only 99 lines-of-code within Geth to expose sufficient internal EVM state to support the prover. This demonstrates the architecture's simplicity and feasibility of extending to other L2 client programs.
- We integrate these components to implement Specular, the first optimistic rollup to achieve (1) a minimal TCB, improving security, auditability and upgradeability, (2) support for permissionless participation of multiple Ethereum client programs, enabling client diversity, and (3) sufficiently efficient dispute resolution.

2. Background

2.1. Ethereum

A blockchain is a decentralized append-only ledger, managed by a network of mutually distrusting parties. *Ethereum* is a permissionless, programmable blockchain that exposes a general-purpose state machine with a quasi-Turing complete[2] programming language. Users can deploy smart contracts on Ethereum to program arbitrary stateful logic into the blockchain. At the request of a signed user transaction, a contract call is atomically executed in the Ethereum Virtual Machine (EVM) by the blockchain network.

2.2. Ethereum Virtual Machine

This subsection focuses on the EVM state relevant to transaction execution, as semi-formally specified by the Ethereum Yellow Paper [1]. The EVM is a virtual stack machine that defines how bytecode instructions alter the Ethereum state. It has a volatile memory represented by a word-addressed byte array and a non-volatile storage, represented by a word-addressed word array. Both memory and storage are zero-initialized at all locations. EVM bytecode is stored in virtual read-only memory, accessible only through a specialized instruction. The EVM state (both volatile and non-volatile) is split across the EVM world state, machine state, accrued substate and environment information. We summarize each of these below. A full definition of this state can be found in the Ethereum Yellow Paper [1]. The semantics and gas cost corresponding to each instruction which mutates the state can be found there in Appendix H.

The world state $\boldsymbol{\sigma}$ is a mapping between addresses and account states, stored in a Merkle Patricia tree (trie). Each account $\boldsymbol{\sigma}[a]$, identified by its address a, is comprised of an intrinsic monetary $balance$ and transaction count $nonce$. An account is also optionally associated with storage state and EVM code through a `storageRoot` (256-bit hash of storage MPT root) and `codeHash` (hash of bytecode stored in a separate state database) respectively. All fields are mutable except `codeHash`, which is immutable (more specifically, write-once on contract creation). If an account has no associated bytecode, it represents and can be controlled by an external entity; otherwise it represents an autonomous object that can be invoked.

The machine state $\boldsymbol{\mu}$ of the current messsage-call or contract creation is a 6-tuple $(g, pc, \boldsymbol{m}, i, \boldsymbol{s}, \boldsymbol{o})$ of the remaining

2. Transaction size is bounded by a resource parameter, the gas limit.

gas available g, program counter $pc \in \mathbb{N}_{256}$, memory contents $\boldsymbol{m}$, active number of words in memory i, stack contents $\boldsymbol{s}$, and the return data from the previous call $\boldsymbol{o}$[3].

The execution environment information is a tuple of read-only data $I \coloneqq (I_a, I_o, I_p, I_{\boldsymbol{d}}, I_s, I_v, I_{\boldsymbol{b}}, I_H, I_e, I_w)$ that can be accessed by specific instructions. This includes the account address of the executing bytecode I_a, the transaction sender address I_o, gas price I_p, input data $I_{\boldsymbol{d}}$, invoker account address I_s, monetary value I_v, bytecode $I_{\boldsymbol{b}}$, current block header I_H, call depth I_e, and state modification permission bit I_w.

The accrued transaction substate A is state accrued during a transaction's execution and is used to update EVM state immediately post-transaction. This is defined as a tuple $A \coloneqq (A_s, A_{\boldsymbol{l}}, A_{\boldsymbol{t}}, A_r, A_{\boldsymbol{a}}, A_{\boldsymbol{K}})$, containing the self-destruct set A_s, series of logs emitted $A_{\boldsymbol{l}}$, set of touched accounts $A_{\boldsymbol{t}}$, refund balance A_r, set of accessed account addresses $A_{\boldsymbol{a}}$ and set of accessed storage keys $A_{\boldsymbol{k}}$.

Finally, the 4-tuple of the above-defined state $(\boldsymbol{\sigma}, \boldsymbol{\mu}, I, A)$ comprises the complete EVM state that can be read from or written to by a bytecode instruction.

2.3. Refereed delegation

Refereed delegation of computation (RDoC) [23], [24] consists of a family of protocols that allow a resource-bound client to efficiently and verifiably compute a function by delegating it to multiple untrusted servers, provided at least a single server is honest. Most notably, Canetti et al. introduce a computationally sound interactive protocol, based on any collision-resistant hash function. The protocol is *general-purpose* (supports any efficiently computable, deterministic function) and *full-information* (requires no private state).

The protocol can be summarized as follows; suppose a client delegates the computation of a function to two non-colluding servers. If the servers unanimously agree on a result, the client accepts it immediately—since by assumption, one server is honest. If the servers disagree, it initiates a bisection protocol with a logarithmic number of rounds to search for inconsistencies between the intermediate states of the servers' delegated computation. In each round, the servers send the client commitments to their intermediate states at the computation step requested, generated using the hash function (to avoid sending the entire state).

On identifying the inconsistency at the level of a single step (e.g. an instruction), the client determines which party is dishonest by (1) requesting the initial state to reveal the commitment previously received and agreed upon by both servers; and (2) locally evaluating the step, accepting the result claimed by the honest server. This protocol can be extended trivially to an n-server disagreement, through pairwise execution.

In cases of unanimous agreement, there is no additional computational overhead for both the servers and the client. In disagreement, the overhead is poly-logarithmic in the size of the computation.

3. This field is omitted from the definition of $\boldsymbol{\mu}$ in the EVM specification [1]—possibly unintentionally—but can be found in Appendix H where the semantics of `RETURNDATASIZE` and `RETURNDATACOPY` are described.

2.4. Optimistic Rollups

Optimistic rollups (ORUs) extend RDoC protocols to the permissionless blockchain setting, where the trusted L1 blockchain can be considered a resource-bound client and L2 parties the more powerful servers. RDoC protocols are well-suited to blockchains, where all execution is deterministic, and the L1 has no private state. However, because there is no specific pre-determined committee of non-colluding servers that can be relied on by L1 (as is the case in RDoC protocols), ORUs allow *any* party to participate at L2 as a validator by posting, attesting to or disputing state commitments. This policy is programmed into the ORU's *bridge*, which resides on L1 and is typically implemented as a smart contract, or contracts [2].

Users submit transactions directly to L2. ORUs decouple transaction sequencing from state computation; a sequencer decides on transaction ordering off-chain before posting a batch to L1, while validators (fulfilling the RDoC server role) read and execute these transactions in an off-chain virtual machine (L2VM). Many ORUs also provide a fall-back on-chain mechanism that provides censorship resistance, to prevent or disincentivize sequencers from censoring specific users or transactions [2].

Upon executing a batch of sequenced transactions, validators post and attest to a cryptographic commitment of the resulting new state on L1, asserting its validity. Any party can validate and attest to a state commitment (also referred to as a *disputable assertion*, or DA); and when necessary, any party can initiate a dispute on L1 within a pre-configured *dispute period* to argue its invalidity and trigger its (eventual) rejection. The length of the dispute period is a system parameter and depends on several factors, including the maximum DA gas size, with existing protocols (including Arbitrum) conservatively affording a week [25].

As in the case of RDoC, the validity of ORU state updates is therefore guaranteed under a single honest party assumption. That is, as long as a single honest party exists in the network to follow the protocol and initiate disputes as necessary, an invalid state will not be confirmed on-chain. We elaborate on the dispute resolution protocol used by this work, which is adapted from [16], [26], in Appendix A; the focus of our work is on the final round (local step evaluation), which is described in detail in Section 4.

EVM reuse. Reuse of the EVM and other Ethereum abstractions at L2 has conventionally been described in two forms: (1) EVM compatibility and (2) EVM equivalence.

A rollup is described as *EVM-compatible* if it allows Ethereum smart contracts to be ported—with relatively low developer effort—to compile and execute on its L2VM. Notably, EVM compatibility does not imply a strict implementation of EVM semantics. For example, Arbitrum supports Ethereum contracts written in the high-level language Solidity (with minor, if any, changes) but uses a different ISA, that of the AVM, under-the-hood [16].

Recent efforts by Arbitrum (in Nitro) [5] and Optimism (in Cannon) [4] aim to achieve a higher degree of EVM reuse to reduce the remaining burden of incompatibility on

Type	Properties							
	Ecosystem compatibility‡	L1 client reuse	One-step proof target ISA	Level of equivalence	TCB	Upgrade frequency††	Upgrade transparency††	Intrinsic client diversity
EVM-compatible†	Limited	✗	Custom VM	Solidity	Large	Frequent	✗	✗
EVM-equivalent	Full	✓	Lower-level VM	Client binary	Large	Frequent	✗	✗
EVM-native	Full	✓	EVM	EVM spec	Minimal	Rare	✓	✓

TABLE 2: Comparison of types of EVM reuse. † EVM-compatibility is a misnomer and does not imply semantic equivalence with the EVM. ‡ Ecosystem compatibility refers to both Ethereum smart contracts and the Ethereum toolchain broadly. †† These properties refer to the frequency and transparency of upgrading the TCB, including the L1 verifier.

users and developers, and benefit from prior engineering efforts at L1 (specifically on Geth) for improved security and performance. This has been described as *EVM-equivalence* [21], providing complete compliance with the semi-formal Ethereum protocol specification [1]. An EVM-equivalent rollup enables a compiled Ethereum contract to be redeployed as is, without recompilation.

To support one-step proofs, the Geth EVM Golang source is compiled to the ISA of a lower-level target VM (MIPS by Cannon, WASM by Nitro). A vector commitment is generated over the resulting binary, where each element is an instruction. The commitment is posted on-chain, allowing the verifier to enforce the binary's execution at the granularity of the lower-level ISA. Crucially, this approach does not directly enforce EVM semantics. Instead, it enforces the execution of a *specific* client binary. In the next section, we enumerate over the challenges that arise due to this design and how an EVM-native IFP resolves them.

3. Our Approach: EVM-native IFP

To solve the aforementioned problems, we propose enforcing EVM semantics (as defined in the semi-formal specification [1]) explicitly on-chain, at the level of a single EVM instruction. We outline the advantages of the EVM-native approach below in comparison with the lower-level VM approach.

3.1. Client diversity

While the Ethereum network is run by a diverse set of clients including Geth, Besu and Erigon—albeit presently with insufficient adoption [27]—no current ORU supports any form of client diversity whatsoever. This allows invalid ORU state transitions induced by software bugs or vulnerabilities to slip by undetected and undisputed.

Extrinsic client diversity. Optimism's developers have proposed one potential solution [14], which extends the lower-level VM approach to the multi-client setting by including an on-chain binary commitment for each of a set of whitelisted client programs. Validators must be prepared to participate in challenge phases involving any one of them, only winning a dispute if they manage to win a threshold majority of the challenge phases invoked overall. This approach only provides *extrinsic* client diversity—each client has its own exclusive proof system associated with it. While this may offer some limited defense against monoculture failures, it introduces other issues.

First, use of a client program in dispute participation is *permissioned*. That is, a client program's compiled binary must be committed to on-chain as a whitelisted program for it to be available to use in dispute resolution. Moreover, since the binary commitment changes from version-to-version, the ORU's governance process would also on the critical path for *any upgrade to any client program* (or even its compilation toolchain). Ideally, a client program can be used and upgraded without going through the ORU's on-chain governance.

Second, there is a strong trust assumption that a threshold majority of client programs implement the EVM correctly. This can be interpreted as K-of-N-version, or N-of-N-version programming [28]. As we note in the next subsection, this is particularly problematic because client programs are difficult to audit. Ideally, it should be sufficient to require only a single client program to implement the EVM correctly. This is equivalent to using classical N-version programming [7].

Third, validators have no choice but to be prepared to run every whitelisted client program to resolve a dispute. Not only is this expensive and slow due to the additional redundancy in the dispute process, it is also more operationally complex, as a validator must learn how to configure and run each program. Moreover, the operational complexity grows with the number of clients, placing practical limitations on how many client programs may be included.

Intrinsic client diversity. The EVM-native approach provides *intrinsic* client diversity, because the on-chain verifier is agnostic to the client program. All clients interact using the same proof system. Any program that supports EVM semantics can be utilized permissionlessly, and *only one* must implement the correct semantics for faults to be detectable and disputed. That is, unlike in prior work, this enables N-version programming [7]. Additionally, validators are afforded a choice in running their client program of choice, providing simpler devops in practice.

An EVM-native ORU therefore provides support for trust-minimized, permissionless client diversity. While both approaches enable resolution of maliciously injected state transitions (e.g. caused by invalid transactions), only the EVM-native approach enables robust detection and resolution of bugs and vulnerabilities.

3.2. TCB Trustworthiness

The TCB of an ORU must be auditable, and ideally feasible to formally verify to ensure its trustworthiness.

Existing ORUs fall well short of this objective.

Auditability. In the lower-level VM approach, because the on-chain verifier enforces the execution of the client program at the target ISA instruction-level (rather than higher-level EVM semantics), inspecting the verifier alone is not sufficient to determine the enforced semantics. The enforced semantics of the L2VM are determined by the whitelisted client binary. It is impossible to determine whether these semantics are equivalent to EVM semantics a priori without auditing the entire client program and compiler that produced the binary. Moreover, with extrinsic client diversity, this problem is exacerbated further; semantics are determined by *majority consensus* (manifesting through the dispute mechanism)—requiring extensive audits for *all*, or at least a majority of, client programs and compilers.

The TCB therefore includes not only the verifier, but also the client programs, compiler toolchains and binary commitment generator. The bloated size and complexity of the TCB magnifies the risk of vulnerabilities that can impact the security of the ORU, and commensurately increases auditing overheads. We also note that it is not possible to completely piggyback off of security audits conducted by the original client program maintainers (e.g. for Geth), because the program must be custom-tailored to support one-step proof generation. For example, since the on-chain verifier cannot have access to the Ethereum database, which stores blocks, transactions, and states like accounts and storage, a preimage oracle component must be used to allow instructions to query the database.

In the EVM-native approach, EVM semantics are explicitly enforced by the verifier. With trust-minimized, permissionless client diversity, the overall security of the ORU does not rely significantly on the correctness of any individual client program. The TCB of an EVM-native ORU therefore includes only the on-chain verifier. This reduction of scope improves auditability and fulfills a necessary condition for formal verification with respect to a formal EVM specification, as we touch on below.

Formal verification feasibility. Formal verification can allow us to verify the correctness of the on-chain verifier against a formal specification. In particular, KEVM [11] is an executable formal specification of the EVM implemented using the K-Framework [29] that supports the formal verification of Solidity smart contracts. KEVM is therefore complementary to our work because it provides both a formal specification and a suite of verification tools that can be directly applied to a verifier contract implemented in Solidity. A formally verified one-step proof verifier guarantees that EVM semantics are correctly enforced on-chain, strengthening the trustworthiness of the TCB.

On the other hand, it is infeasible to formally verify the TCB of ORUs that take the lower-level VM approach. While it *is* possible to formally verify a MIPS verifier, this does not provide any assurances regarding the enforcement of EVM semantics, as described earlier. To achieve such assurances, Geth itself—along with its compilation toolchain—must be formally verified; however, given the unbounded and concurrent nature of these programs, this is infeasible.

3.3. Upgrades

Frequency. While Ethereum client programs are frequently upgraded, the Ethereum protocol itself tends to hard-fork roughly only a couple times a year [12]. Hard-forks that actually modify EVM semantics are even less common, at around once-a-year. Moreover, many of these changes (e.g. at the consensus layer) do not affect execution semantics. As a result, the L1 contract and TCB of an EVM-native ORU needs relatively infrequent upgrades, compared to ORUs taking the lower-level VM approach. Comparatively, the lower-level VM approach requires an L1 contract upgrade every time the client upgrades, since the changes must be reflected in a new binary commitment on-chain.

Furthermore, we expect the Ethereum protocol and EVM specification to eventually stabilize. Therefore, in the long-term, the frequency of upgrades to the TCB of EVM-native ORUs will decrease, tending to zero. On the other hand, individual client programs will likely continue to commonly experience upgrades that intend to patch vulnerabilities, fix bugs, and generally improve performance—triggering frequently necessary upgrades in the lower-level VM approach.

Additionally, an EVM-native ORU's source code can stabilize in tandem with Ethereum. This is because Specular's developers can forfeit L1 contract upgrade control without forfeiting the ability to ship vulnerability patches, bug-fixes and performance improvements. The lower-level VM approach cannot offer this, due to the inherent coupling of the client binary and on-chain verifier. We therefore argue that the safest and most practical path to relinquishing upgrade keys is through an EVM-native ORU design.

Transparency. The size and complexity of the TCB in the lower-level VM approach results in an opaque upgrade process. For example, client programs are upgraded in a less transparent manner than the Ethereum specification, which undergoes a deliberate, public and peer-reviewed request for proposal process [12]. Upgrades to components of the toolchain specific to the ORU, such as the Golang-to-ISA compiler, are even less transparent still.

In our approach, there is a clear separation between verification of semantics and the client program implementing those semantics. Therefore, it is easier to discern whether or not an upgrade can potentially affect the interpretation of semantics—auditors need only look at the diff in the source code in the L1 contracts.

4. Our One-step Proof System

4.1. Requirements

The one-step proof (OSP) aims to convince the verifier that given an initial EVM state—revealed and proven consistent against a commitment—executing the current instruction will result in a transition to the final EVM state the prover has committed to. The system must address the following requirements.

1. **EVM-native.** The OSP attests to the validity of a state transition at the granularity of a single EVM instruction. All EVM instructions and inter-transaction operations are supported.

2. **Specification-compatible.** The OSP can be constructed exclusively from state specified by and accessible from the EVM (i.e. without relying on knowledge of a specific EVM client implementation).
3. **Efficient.** The OSP generation procedure runs quasilinearly in the size of EVM state b; the size of the proof is logarithmic in b, linear in contract size and linear in bytes accessed. Concretely, the verification cost of a OSP should fit within a single on-chain transaction (on Ethereum, this is 30 million gas).

The key challenge is to design a OSP system for the EVM that is sufficiently efficient to verify on-chain. This requires commitment schemes for the stack, memory and storage and other EVM states that support efficient partial reveals and updates. We use simple and well-studied techniques, to maintain a small and auditable TCB.

4.2. State Definitions

In this subsection we define the EVM state we generate proofs over. We borrow formalism previously used to describe the EVM (see Section 2) wherever convenient.

The *one-step state* (OSS) is the state between EVM transaction execution steps. The type of EVM transaction execution steps includes EVM instruction execution, transaction initiation/finalization, and block initiation/finalization. The OSS has three forms: *intra-transaction state*, *inter-transaction state*, and *block state*.

Intra-transaction state. The *intra-transaction state* ω of the EVM represents states between EVM instruction executions. It is directly constructed from the full EVM execution state $(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)$. It contains every state field modifiable by EVM opcodes, including gas, stack, memory, and world state (a full definition of the intra-transaction state and the commitment to it, $H_{\texttt{OSS}}$, is included in Section B.1).

For efficient verification, the commitment $H_{\texttt{OSS}}$ does not directly hash the contents of fields that have inner structures (referred as *components*), such as the stack, memory, and world state; instead, it hashes on separate commitments designed for each component. This allows us to create the *state proof* (described in Section 4.4.1) without directly including the contents of each component. A separate proof is submitted to prove the validity of the state transition in the component. Since opcodes do not use every component (e.g. `ADD` doesn't access or modify the memory or the world state), the OSP is modularly designed, providing only the proofs necessary.

Inter-transaction state. To encode EVM behavior that takes place between the consecutive execution of transactions, we define a special type of one-step state, the *inter-transaction state* $\omega_{\texttt{int}}$. The inter-transaction state lies between the execution of two transactions, representing the finalized state after the execution of the first. See Section B.1 for the full definition of the intra-transaction state and its commitment.

Block state. Similarly, to encode EVM behavior that takes place between the consecutive execution of blocks, we define another special type of one-step state, the *block state* $\omega_{\texttt{b}}$. The block state lies between two blocks, representing the finalized state of the first. See Section B.1 for the full definition of the block state and its commitment.

During normal execution, only the $H_{\texttt{OSS}}$ of the block state is used as the commitment and posted on-chain. The validators can compute the $H_{\texttt{OSS}}$ of the block state from execution results of Ethereum blocks with negligible impact to normal execution. The validators do not construct any inter-/intra-transaction state, or compute the $H_{\texttt{OSS}}$ of any inter-/intra-transaction state during any point of the normal execution. Only during a dispute does the validator construct inter-/intra-transaction states by re-executing the transaction.

4.3. State Transitions

For a particular transaction T_i, suppose the EVM execution has n steps; the complete state transitions of that transaction can be represented as $\omega_{\texttt{int}}^{(T_{i-1})} \to \omega_1 \to \cdots \to \omega_n \to \omega_{\texttt{int}}^{(T_i)}$, where $\omega_{\texttt{int}}^{(T_{i-1})}$ is the inter-transaction state before T_i and $\omega_{\texttt{int}}^{(T_i)}$ is the inter-transaction state after T_i.

For two consecutive transactions T_i and T_{i+1}, the state transition between them is $\omega_n^{(T_i)} \to \omega_{\texttt{int}}^{(T_i)} \to \omega_1^{(T_{i+1})}$, where $\omega_n^{(T_i)}$ is the last intra-transaction state of T_i and $\omega_1^{(T_{i+1})}$ is the first intra-transaction state of T_{i+1}.

If the transaction T_i is a regular transaction (i.e. it only transfers value between externally-owned accounts but does not invoke any smart contract creation or execution), the state transition is simply $\omega_{\texttt{int}}^{(T_{i-1})} \to \omega_{\texttt{int}}^{(T_i)}$.

If the transaction T is the first transaction in the block B_i, before state transitions of T, there is an additional state transition $\omega_{\texttt{b}}^{(B_{i-1})} \to \omega_{\texttt{int}}^{(B_i)}$ for block initiation, where $\omega_{\texttt{b}}^{(B_{i-1})}$ is the block state of the previous block of B_i and $\omega_{\texttt{int}}^{(B_i)}$ is the inter-transaction state before T.

If the transaction T is the last transaction in the block B_i, after state transitions of T, there is an additional state transition $\omega_{\texttt{int}}^{(T)} \to \omega_{\texttt{b}}^{(B_i)}$, where $\omega_{\texttt{int}}^{(T)}$ is the inter-transaction state after T and $\omega_{\texttt{b}}^{(B_i)}$ is the block state of block B_i.

We discuss the validity of each type of state transition in Section B.2.

4.4. Proof System

Suppose that upon executing the current instruction $I_{\boldsymbol{b}}[\boldsymbol{\mu}_{\texttt{pc}}]$ or an inter-transaction/block step, the EVM transitions from an initial OSS ω to a new OSS $\omega \to \omega'$, where $\omega, \omega' \in \Omega$, the set of all valid one-step states. Let $h := H_{\texttt{OSS}}(\omega)$, $h' := H_{\texttt{OSS}}(\omega')$. Given h and h', the OSP must convince a verifier that the transition $\omega \to \omega'$ (equivalently, the transition from $h \to h'$) is valid.

More formally, the OSP system is a tuple (ω, ω', P, V), where P is the OSP generator, and V is the OSP verifier. P is defined as $\pi := P(\omega, \omega')$. π is a one-step proof of the transition $\omega \to \omega'$. V is defined as $V(C, h, h', \pi)$ which outputs `ACCEPT` if π proves the validity of the transition of the states corresponding to the hashes $\omega \to \omega'$ in one step under verification context C, or `REJECT` otherwise (as illustrated in Figure 3). C contains the calldata of transactions that were posted on L1 to provide the information of transactions, such

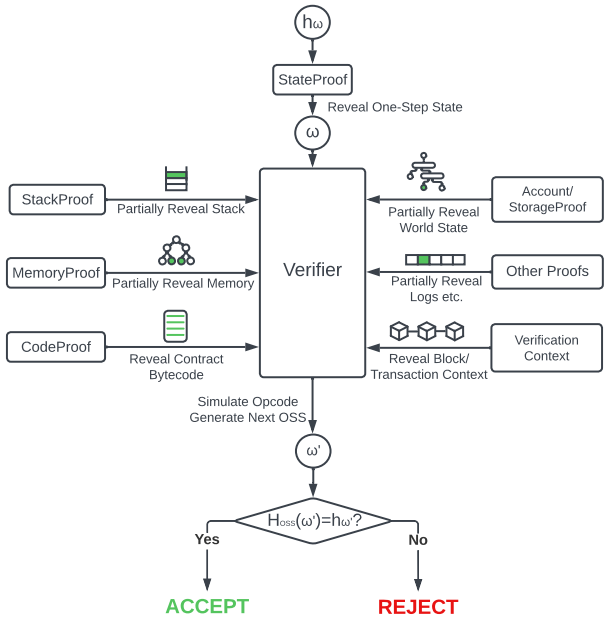


Figure 3: One-step proof verification. An illustration of the flow of inputs and outputs within the one-step proof verifier.

as the sender and recipient, or block contexts associated with transactions like timestamp and block number.

An OSP π is a composition of proofs, each of which proves the validity of a part of the transition $\omega \rightarrow \omega'$. This proof establishes that (1) ω is consistent with h, (2) performing one-step of state transition while in state ω yields ω', and (3) ω' is consistent with h'.

The types of proofs that π consists of depend on the type of the state transition (for state transitions between intra-transaction states, also the opcode of the current instruction) and consequently which EVM data structures are read from or written to (see Table 3 for a summary). Specifically, each type of proof corresponds to the consistency of the OSS and validity of its transition, or that of a component of the OSS (i.e. that of the stack, memory, or world state, etc.). We introduce each type of proof below.

4.4.1. Preliminary proofs

We provide the following proofs for *all* instructions in state transitions between intra-transaction states.

First, a *state* proof establishes the consistency of the OSS, given the hash of OSS. It consists of either the field content of the OSS if that field is a *property* (e.g. execution depth), or the witness of the field if that field is a *component*, i.e., has inner structures. Specifically, we include the OSS hash for the last depth state, the size and stack hash (described in Section 4.4.2) for the stack, the size and the merkle root (described in Section 4.4.3) for the memory, input data, and return data, and the MPT root (described in Section 4.4.4) for the world state, along with witnesses of other components. The verifier is able to compute the hash of OSS by the state proof only, without knowing the entire contents of the OSS. The verifier expects π to always contain the state proof for ω. It uses the state proof to derive ω' by simulating the current opcode, given the support by other types of proofs in π for information of components, and checks if ω' is consistent with h'.

Second, an *opcode* proof attests to the consistency of the next instruction to be executed (at offset $\boldsymbol{\mu}'_{\mathrm{pc}}$). In the current Specular design, the opcode proof naively includes the entire contract bytecode. The verifier can then simply verify the opcode proof by computing the hash of the provided bytecode and comparing against the code hash. The cost of providing and hashing the contract bytecode is still sufficiently efficient since in practice, contract sizes are sufficiently limited in Ethereum for the proof to fit comfortably within a single transaction; however, it is preferable—and indeed feasible to use off-the-shelf techniques—to reduce the opcode proof size significantly further. We propose using a succinct proof to further reduce the proof size and briefly comment on this in Section 7.2.

4.4.2. Stack

The *stack* proof attests to the validity of the state transition of the EVM stack $\boldsymbol{\mu_s} \rightarrow \boldsymbol{\mu'_s}$. We define $H_{\mathtt{stack}}$ as a hash chain over the elements of a stack $\boldsymbol{\mu_s} := (\boldsymbol{\mu_s}[i] \mid 0 \leq i < n)$ of size n, where $\boldsymbol{\mu_s}[i]$ denotes the i-th element of the stack.

$$H_{\mathtt{stack}}(\boldsymbol{\mu_s}) = \begin{cases} \mathtt{EMPTYHASH} & \text{if } n = 0 \\ \mathtt{KEC}(H_{\mathtt{stack}}(\boldsymbol{\mu_s}[0, n-1]) \,||\, \boldsymbol{\mu_s}[n-1]) & \text{if } n > 0 \end{cases}$$

where $\boldsymbol{\mu_s}[i, j]$ denotes the subsequence $(\boldsymbol{\mu_s}[k] \mid i \leq k < j)$ and $\mathtt{KEC}$ denotes the Keccak-256 cryptographic hash function [30]. Let δ be the number of elements to be popped from the stack and α the number of elements subsequently pushed by $I_{\boldsymbol{b}}[\boldsymbol{\mu}_{\mathrm{pc}}]$. We denote h as the hashes of the initial stack states, $h_k^{\mathtt{POP}}$ as an intermediate hash of the stack computed after $\delta - k$ pops ($0 \leq k \leq \delta$), and $h_k^{\mathtt{PUSH}}$ as an intermediate hash of the stack computed after δ pops and k pushes ($0 \leq k \leq \alpha$).

Then, the stack proof is the tuple (h, h_0, p), where h_0 is the intermediate stack hash, and $p := \boldsymbol{\mu_s}[|\boldsymbol{\mu_s}| - \delta, |\boldsymbol{\mu_s}|]$, is the subsequence of the δ top elements popped, where $|\boldsymbol{\mu_s}|$ is the size of the stack.

Given the proof, the verifier computes $h_k^{\mathtt{POP}} := \mathtt{KEC}\left(h_{k-1}^{\mathtt{POP}} || p[k]\right) \; \forall \; 0 < k \leq \delta$, where $h_0^{\mathtt{POP}} = h_0$ and $p[i]$ is the i-th element of p, and checks $h = h_\delta^{\mathtt{POP}}$ to verify the membership of the popped elements. The verifier then simulates the execution of $I_{\boldsymbol{b}}[\boldsymbol{\mu}_{\mathrm{pc}}]$ to compute the sequence q of α elements to push onto the stack. Finally, it computes $h_k^{\mathtt{PUSH}} := \mathtt{KEC}(h_{k-1}^{\mathtt{PUSH}} || q[k]) \; \forall \; 0 < k \leq \alpha$, where $h_0^{\mathtt{PUSH}} = h_0$ and $q[i]$ is the i-th element of q, to get $h_\alpha^{\mathtt{PUSH}}$, the hash of the post-transition stack state. This value is ultimately used to verify the final state proof, confirming the complete post-transition OSS ω' is correct.

4.4.3. Memory

Since the EVM's memory $\boldsymbol{\mu_m}$ is a form of virtual RAM, we must utilize a commitment that supports efficient random-access partial reveals and updates. An obvious choice is to use a Merkle tree over the EVM memory space. $H_{\texttt{OSS}}$ commits to a Merkle tree root computed over the entire contents of the EVM memory space $\boldsymbol{\mu_m}$, where each leaf i is $\texttt{KEC}(\boldsymbol{\mu_m}[32i, 32i + 32])$. Specifically, we use a variant of Merkle tree proof implemented by [31] which supports efficient multi-proofs and append-proofs.

The *memory-read* proof (i, c, p, r) attests to the consistency of the byte array c starting at offset i of $\boldsymbol{\mu_m}$, with respect to memory Merkle root r. The Merkle proof p can be either a single-proof or a multi-proof, depending on the length of c. A multi-proof reveals and attests to the consistency of multiple Merkle tree leaves succinctly.

Likewise, the *memory-write* proof attests to the validity of a write to memory. That is, it proves the validity of the new content at the location written to *and* that the contents at no other location were changed. This proof is of the form (i, c, c', p, r, r'), where r' is the new Merkle root after the byte array c' overwrites c at the contiguous location starting at offset i of $\boldsymbol{\mu_m}$. The verifier first verifies (i, c, p, r) as performed in the memory-read proof to verify p is a correct proof of consistency of c in r. It then performs memory writes based on (i, c', p) to calculate the new Merkle root and compares with r', the new memory Merkle root provided in the proof. This proves that only the memory starting at location i was modified.

The memory-read/write proof are also applicable to the calldata $I_{\boldsymbol{d}}$ and return data $\boldsymbol{\mu_o}$ of EVM since they act as read-only memory during transaction execution.

4.4.4. World state

Reads. Recall that the account object and its storage are both encoded as Merkle Patricia trees (MPTs). The *account-read* proof attests to the consistency of the account state $\boldsymbol{\sigma}[a]$ associated with address a, as read by an instruction. The MPT proof can be used to compute and verify the root as well as provide information corresponding to each account field (`nonce`, `balance`, `storageRoot`, `codeHash`).

Likewise, the *storage-read* proof attests to the consistency of a value read from a storage slot in the MPT with root $\boldsymbol{\sigma}[a]_s$ (assuming the consistency of $\boldsymbol{\sigma}[a]$), as read by an instruction. It includes the content of the storage slot, with its MPT proof.

The *code* proof similarly attests to the consistency of a contiguous sequence of instructions in an account's bytecode. Similarly as in the case of the opcode proof, the code proof naively consists of the entire bytecode. The verifier can verify the consistency of the bytecode by computing the code hash from the code proof and comparing it to the code hash stored in the contract account.

Writes. A nice property of MPT proofs is that it allows the verifier to perform a single write (including update, insertion, and deletion) on the proven path and recalculate the trie root after the write. This is possible because the MPT proof simply comprises all the relevant trie nodes along the proven path. Therefore, the verifier can derive the trie root after the account write with a *account-read* proof and the updated account, which is essentially the *account-write* proof. Similarly, the *storage-write* proof includes a storage-read proof for the content during initial state $\boldsymbol{\sigma}$, the new content to be written to the storage slot, and an account-read proof of the account where the storage belongs.

4.4.5. Other components

There are some additional types of components inside the intra-transaction state, most of which are used for proving specific opcodes. For example, for `LOG` opcodes, we accumulate all the logs emitted using a hash chain for the receipt calculation; for `SELFDESTRUCT` opcode we accumulate all the accounts that are self-destructed for transaction finalization; for `BLOCKHASH` opcode we use the Merkle proof of the block hash tree to attest to the consistency of block hashes of previous blocks.

4.4.6. Inter-transaction and block

The *inter-transaction* proofs attest to the validity of the transaction initiation and finalization steps. Depending on the type of transaction, the inter-transaction proof may include different types of proofs and additional information. For example, the *transaction initiation* proof always includes an *account-read* proof of the sender and the contract to be executed, if the transaction is not regular. Similarly, an *account-write* proof is required for the *transaction finalization* proof to attest to the remaining gas refund, and code changes if the transaction is a contract creation.

The *block* proof attests to the validity of the block initiation and finalization steps. The *block-initiation* proof simply reveals the field of the block state to be proven because there is no state change in the block initiation step. The *block-finalization* proof reveals necessary state, such as the transaction trie root and receipt trie root for the verifier to reconstruct the block header. It also provides the Merkle proof of the block hash tree, so that the verifier can verify the block hash update.

4.4.7. Verification

The OSP verifier verifies the consistency of the initial OSS ω, simulates the execution of the current EVM instruction—given sufficient context by the prover to do so correctly—and depending on the computed hash of the resultant final OSS $H_{\texttt{OSS}}(\omega')$, decides whether to accept or reject π. The overall procedure run by the verifier is as follows:

1. Verify that the hash of the state proof (defined in Section 4.4.1) for ω is equal to h. If not, `REJECT`;
2. If the state is inter-transaction state or block state, verify the validity of the state transition by inter-transaction proof or block proof (defined in Section 4.4.6) and `ACCEPT` if correct, `REJECT` otherwise;
3. Verify the correctness of opcode proof (defined in Section 4.4.1) for the opcode of the next instruction. If it is incorrect, `REJECT`;

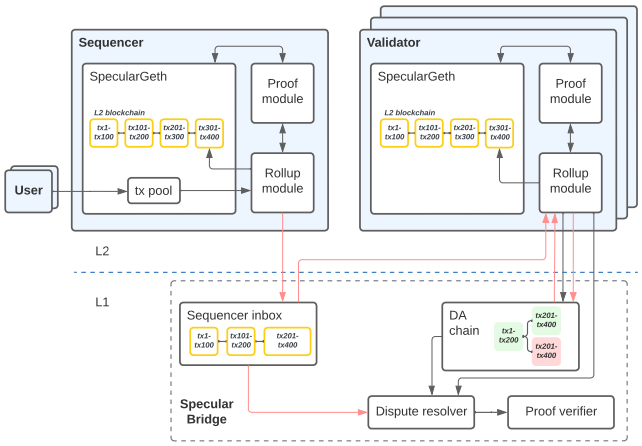


Figure 4: Specular architecture. The sequencer and validator node implementations are simple wrappers around (modified) Geth. Gray arrows represent control flow and pink arrows represent data flow. A sequencer posts data to the sequencer inbox. Validators read this data, and post, attest to or dispute DAs on the DA chain contract.

4. Verify the correctness of remaining proofs in π based on the type of proofs, as described in the previous subsections separately. If any proof is incorrect, `REJECT`;
5. Simulate the execution of the $I_{\boldsymbol{b}}[\boldsymbol{\mu}_{\mathrm{pc}}]$, given C and π. C and π partially reveal the EVM execution environment to the simulator, which executes $I_{\boldsymbol{b}}[\boldsymbol{\mu}_{\mathrm{pc}}]$ according to its semantics, as illustrated in Figure 3. If the simulation fails due to an unexpected exception or the result differs from π, `REJECT`;
6. Construct the state proof for ω' from the simulation result. Verify that the hash of the state proof for ω' is equal to h'. If not, `REJECT`; otherwise, `ACCEPT`.

Runtime analysis. Suppose during an OSS transition $\omega \rightarrow \omega'$, the sizes of the current world state, memory, input data, return data, and bytecode executing are s_σ, s_m, s_i, s_o, and s_c, respectively. Suppose the sizes of memory, input data, return data and bytecode accesses in bytes during the state transition is n_m, n_i, n_o, and n_c, respectively. Suppose the number of account or storage slot accesses during the state transition is n_σ. The worst-case space complexity of the OSP is $\mathcal{O}(n_\sigma \log s_\sigma + \log s_m + \log s_i + \log s_o + s_c + n_m + n_i + n_o + n_c)$. The worst-case time complexity of OSP verification (or gas cost equivalently) is $\mathcal{O}(n_\sigma \log s_\sigma + n_m \log s_m + n_i \log s_i + n_o \log s_o + n_c + s_c)$.

5. Implementation

In this section, we describe the end-to-end implementation of a new optimistic rollup system, Specular. Specular comprises of a fraud proof system that utilizes the one-step proof scheme outlined in Section 4, an L2 sequencer, L2 validator nodes and an L1 bridge. Following prior work by Optimism [15], we similarly implement Specular's sequencer and validator nodes as wrappers over a Geth fork [6] (which we refer to as SpecGeth), as illustrated in Figure 4 [15]. Note that SpecGeth in Figure 4 corresponds to the L2-adapted Ethereum client component in Figure 2. The concrete difference between SpecGeth and Geth is only 99 lines of additional codes inside Geth core, which exposes select internal EVM state to the proof module. The fraud proof system and rollup services responsible for orchestrating sequencing and validation are implemented as separate modules outside SpecGeth. This modular design enables ease of extension to other Ethereum client implementations.

The L1 bridge contracts are implemented in Solidity `0.8.4`, and closely mirror those of Arbitrum [26]—the key differences being in the structure of the disputable assertion and OSP verifier. The bridge functionality is split across four key contracts: (1) a sequencer inbox, where L2 transaction data is persisted within the calldata on the L1 blockchain; (2) a DA chain, where DAs are created, attested to and disputed; (3) a dispute resolver, which is deployed by the DA chain contract upon dispute initiation to referee multisection; and (4) the OSP verifier, called in the final round.

5.1. Fraud Proof System

We implemented the *proof module* (illustrated in Figure 2), which exposes a set of RPC APIs to validators for DA commitment generation and one-step proof generation. During normal execution, the proof module is able to generate DA commitments by constructing the inter-transaction states directly from the transaction data or receipts. When a challenge starts, the proof module re-executes the transaction that is ultimately disputed, and interacts with the EVM through the `EVMLogger` API, which records sufficient EVM state information in each execution step. This information is then used to construct one-step states for intermediate commitments and to generate the final one-step proof. Our implementation of the proof module itself comprises of about ~5,000 lines of Go code, including ~1,000 lines for OSS construction, ~1,000 lines for OSP definitions and encoding, and ~3,000 lines for OSP generation.

The OSP verifier in the L1 bridge comprises of verifier contracts and library contracts. To avoid exceeding the maximum contract size limit, instead of having one verifier contract that handles all types of one-step proof, we split the verifier into a `VerifierEntry` contract and a set of `Verifier` contracts. The `VerifierEntry` contract dispatches one-step proof verification requests to different proof-specific verifiers. The `Verifier` contract set consists of 8 different verifiers, each of which implements verification of several types of one-step proofs. The library contracts provide common utilities for all verifiers, including one-step proof definition and decoding, verification context, EVM gas metering, and EVM consensus parameters. The verifier is implemented in ~4000 lines of Solidity code in total, including ~2000 line-of-code for library contracts and ~2000 line-of-code for all verifier contracts.

5.2. Specular L2 Client

Sequencer. The sequencer orders and executes transactions locally, and periodically posts transaction batches, along with contextual information (e.g. block number and times-

tamp), to the inbox contract on L1. This data is stored as a part of the L1 transaction `calldata`, and encodes sufficient information for any party to fully reconstruct the rollup state. Users submit transactions via RPC to the Sequencer, which batches transactions in a FIFO manner.

As supported in Arbitrum, the inbox contract provides censorship resistance via a mechanism that allows any user to submit transactions directly to the inbox contract. The contract enforces that the transactions will eventually (not necessarily immediately) be included by the sequencer in a batch within an L1-configured time frame. The sequencer's bridge client listens for these L1-queued transactions and injects them into its transaction pool as required.

Validator. Validators subscribe to transaction batches posted to L1 by the sequencer and inject the transactions into their local SpecGeth instance's blockchain, using the provided batch context.

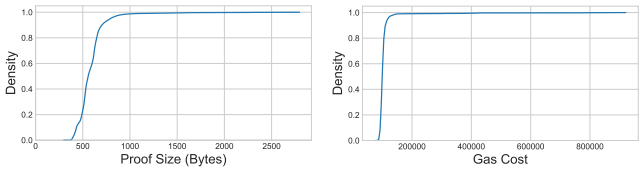
The bridge contract allows staked validators to post their own DA or attest to an existing one, as long as it is a child of a previously attested-to DA or is the last one confirmed. It allows staked validators to initiate a dispute against validators that have attested to sibling branch DAs, if they exists. The concurrently running bridge client is responsible for creating, attesting to and disputing DAs in an event-driven manner, as it listens to events emitted by the bridge. It can be configured to perform any combination of these actions in real-time (note that initiating a dispute requires attesting to a DA first).

On receiving a DA which conflicts with one already posted and validated, the validator can invoke the dispute initiation mechanism in the bridge contract against the creator or any other validator that has attested to it. If the validator ultimately wins the dispute, it can collect its winnings—seized from the counterparty—from the bridge contract. Otherwise, its own staked funds are seized.

On receiving an event marking the start of a dispute against itself, the validator begins its participation in the dispute protocol through the newly created DA-specific dispute resolution contract. The event contains commitments to intermediate states between the final state committed to by the previous DA and that of the current one. The validator replays instructions, and computes and compares commitments using the API of the proof module until the conflicting commitment is found. The validator then generates and posts new commitments to intermediate states between the last agreed upon commitment and the conflicting one. This protocol continues until a one-step disagreement is found, upon which a OSP is generated and submitted to the verifier.

6. Evaluation

In this section, we aim to show that our system's dispute resolution performance is practical and supports sufficiently efficient dispute resolution. We leave performance improvements to future work and discuss low-hanging fruit in Section 7.2. Experiments are performed on a laptop with a AMD Ryzen 9 5900HX @ 3.30 GHz processor, 16 GB memory and Ubuntu Windows Subsystem for Linux.



(a) Proof size CDF **(b)** Gas cost CDF

Figure 5: Proof sizes and verification gas costs for Uniswap v2 transactions (sans contract effects). (a) shows the distribution of one-step proof sizes. (b) shows the distribution of verification gas costs.

To study the performance characteristics of our one-step proof in real world conditions, we carry out disputes on state transitions resulting from interactions with Uniswap V2, a popular decentralized automated market maker. We deployed the Uniswap V2 contract [32], along with 4 ERC-20 token contracts, on Specular. We then generated and executed 100 transactions, each of which is a call to one of the following Uniswap contract functions (sampled uniformly at random) with random parameters: `AddLiquidity`, `RemoveLiquidity`, `SwapTokensForExactTokens`, `SwapExactTokensForTokens`. These functions will interact with the following contracts (with bytecode size in parenthesis): `ERC20` (2.1KB), `UniswapV2Pair` (8.6KB), and `UniswapV2Router02` (17.5KB).

For evaluation purposes only, we construct one-step states and generate one-step proofs for *all* executed state transitions (during normal execution in a real deployment, we do neither). In total, we execute 100 transactions, resulting in a total of 594,583 steps of execution. We construct one-step states for each step for the purpose of this experiment. We then evaluate one-step proof generation and verification on each executed state transition in next sections. Figure 5 illustrates the overall proof size and verification gas cost distributions.

As mentioned in Section 4.4.1, we naively include the entire contract bytecode in the proof, to prove consistency of the bytecode. We plan to replace this approach with a SNARK to mitigate the cost of including the bytecode. Therefore, to better illustrate the actual properties of the one-step proof evaluated, Figures 5, 6 and 7 do not include the effects of contract bytecode (however, the associated text in this Section does). We note that Specular's approach is practical even if the entire contract bytecode is included in the proof—as shown in the next subsections.

One-step proof size. We measure the latency of generating a one-step proof (across ~600,000 steps) to be ~5ms on average. Proof generation therefore has negligible latency.

The average size of the one-step proofs generated—without contract bytecode included—is 552B (min. 331B, max. 2764B). The proof sizes *with* contract bytecode included for opcode and code proofs are an average of 12.3KB (min. 2.63KB and max. 29.7KB). This size depends largely on the contract in which the step occurs. Figure 6 provides a distribution of proof sizes on a per-opcode basis. As a

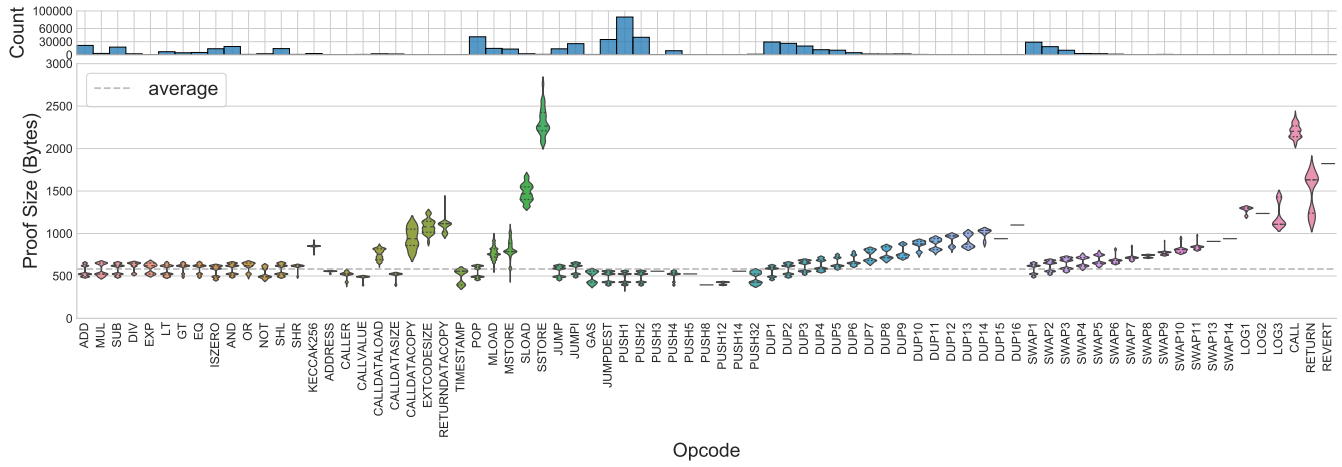


Figure 6: Proof sizes for Uniswap v2 transactions (sans contract effects). Illustrates proof size distribution on a per-opcode basis, along with the distribution of opcodes in the contract.

baseline, Arbitrum claims that the average size of an AVM one-step proof is ~200B, with a maximum of ~500B. However, we expect the proof sizes in both Nitro and Cannon to be significantly larger than that of the AVM, since they generate one-step proofs on lower-level ISAs, which use a flat memory model like that of the EVM (no experimental results have been made public yet by either project).

The EVM, unlike the AVM, is not designed to generate succinct one-step proofs. However, given the rarity of disputes, we find that the sizes of OSPs generated by Specular are practical. The next subsection shows that the overall verification cost is sufficiently low.

One-step proof verification cost. The average gas cost for the verification of the one-step proofs generated—again, ignoring contract size effects—is 104,000 gas (min. 84,000, max. 911,000). For reference, a typical Uniswap v2 swap transaction on Ethereum consumes ~170,000 gas. This is 2-3 orders of magnitude under the Ethereum block limit. Figure 7 provides a distribution of proof sizes on a per-opcode basis.

When contract bytecode is included in one-step proof for either opcode proof or code proof, the gas costs is an average of 609,000 gas (min. 148,000, max. 2,082,000). This increase in gas cost includes the cost for including contract bytecode in calldata, copying it into memory and hashing to verify its consistency. Given that contracts deployed on Ethereum cannot exceed the size limit of 24KB, we estimate that the maximum gas cost in the worst case will not exceed 3,000,000 gas (i.e. only 10% of the Ethereum block gas limit). Thus, the verification cost of our approach in the worst case is still practical even when the full contract bytecode is included in the proof.

Aside from the gas cost introduced by the contract bytecode inclusion, a significant factor in proof verification gas cost is MPT proof verification for opcodes that access the world state, as shown in Figure 7. `SSTORE` requires four MPT proofs, including two for the account and storage (one for the storage slot of the committed state for gas charges and one for the storage slot write). For opcodes that access the memory, the verification cost has large variance because total memory utilized varies significantly (affecting the height of the Merkle tree) and the size of memory accessed by the opcode varies significantly (affecting the cost of Merkle multi-proof verification).

7. Discussion

7.1. Related Work

Existing systems can be categorized as either *optimistic rollups* or *validity rollups* (also colloquially referred to as zk-rollups). We elaborate further on related work in the former category and touch on the relevance of our work in the context of the latter.

Optimistic rollups. Concurrently, there have been ongoing open-source efforts in both the Arbitrum and Optimism communities towards designs that—similar to our work—allow for reuse of an Ethereum client and EVM implementation, while supporting interactive fraud proofs that target a lower-level VM [5], [4]. We have discussed the disadvantages of this approach in previous sections, but touch on notable trade-offs below.

One advantage of the lower-level VM approach over the EVM-native one is the simplicity with which L2 functionality that extends the EVM can be implemented. This is because the on-chain verifier is oblivious to L2 semantics and can enforce the execution of *any* binary committed to (note that this is the same reason such systems hinder client diversity). In fact, this proves useful, for example, to reduce user gas fees via transaction data batch compression, pre-sequencing. The validity of the data compression step can be proven to the verifier using the same IFP system. While extending L2 functionality beyond the EVM is still possible in an ORU utilizing an EVM-native approach, it requires the implementation of a custom on-chain verifier.

Another advantage of this approach is that modifications to the EVM specification do not necessitate corresponding

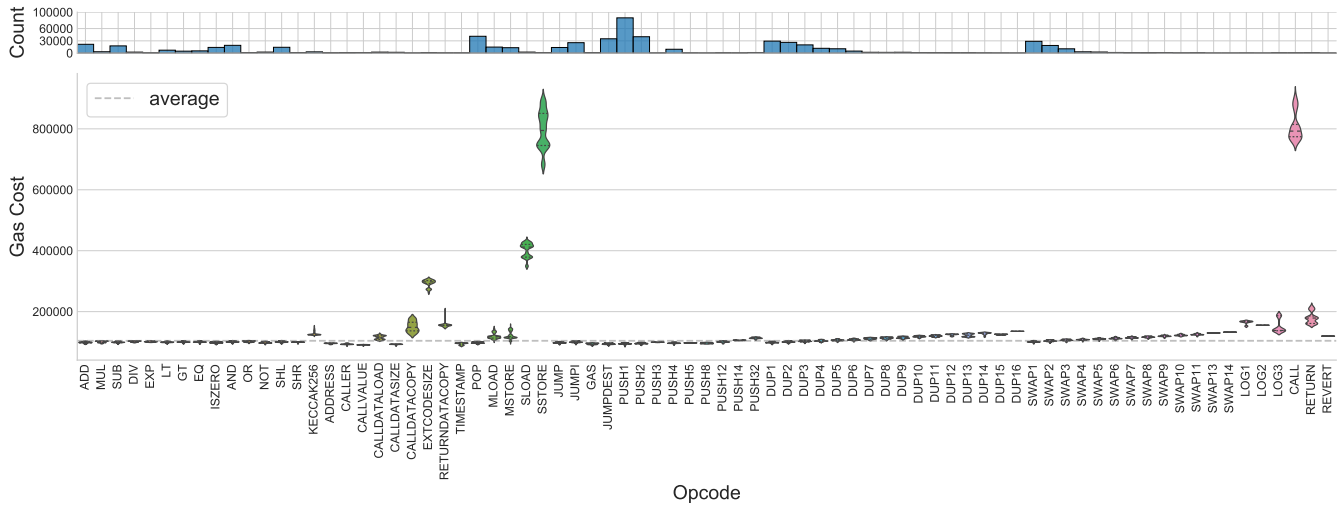


Figure 7: Verification gas costs for Uniswap v2 transactions (sans contract effects). Illustrates gas cost distribution on a per-opcode basis, along with the distribution of opcodes in the contract.

changes in the proof verifier, since the lower-level ISA remains the same. This may reduce some maintenance overhead on L1. However, the commitment to the L2 client binary must still be updated in tandem with any client upgrade—even if the EVM specification does not change. While our approach features a more complex OSP verifier, it requires a bridge upgrade only when the EVM specification itself is upgraded. It is also straightforward to compose the necessary subproofs (defined in Section 4) to do so and can be done in a few lines of Solidity code—both for example, in the case of a new opcode or pre-compiled contract.

Oasis is another blockchain that utilizes ORUs [25]. But unlike Ethereum, it does so by providing native support for fraud proofs within the protocol. Specifically, L2 state disputes are detected by an L1 network-sampled committee. However, disputes are resolved in the slow-path non-interactively by executing all transactions involved.

Validity rollups. In a validity rollup, a state update posted on-chain must be accompanied with a validity proof—typically a succinct non-interactive argument of knowledge (SNARK)—that convinces the on-chain verifier of the correctness of the computation resulting in the new state. The security of a validity rollup relies primarily on the soundness of the SNARK verifier. The L2 client program correctness and prover completeness are only necessary for system liveness; as a result, client diversity is not essential here.

However, this isn't completely free and comes with trade-offs. The TCB constitutes a complex SNARK verifier that uses heavy cryptographic machinery, which can be difficult to understand and audit. Incidentally, this machinery also poses a challenge for popular formal methods tooling, such as SMT solvers. While there has been some progress on making formal verification for simple SNARK systems practical [33], formal proofs of soundness for more complex systems, such as those used by zkEVM systems, are currently out of reach.

7.2. Future Work

Formal verification. We plan to formally verify our L1 verifier against an existing Ethereum formal specification, such as that of KEVM [11] (built on K-framework [29]). This would establish semantic equivalence between our verifier and the Ethereum specification, which has two obvious benefits. It (1) further minimizes trust in Specular's source code; and (2) provides a credible basis for the system maintainers to forfeit upgrade control of L1 rollup contracts, while retaining the ability to ship client program upgrades.
Contract size mitigation. As demonstrated in the evaluation, contract bytecode size and verification dominate the overall one-step proof size and verification cost respectively. We plan to use off-the-shelf zero-knowledge proof methods to avoid the inclusion of contract bytecode in the opcode proof (Section 4.4.1) and code proof (section 4.4.4), thereby reducing the size and verification cost of the one-step proof by an order of magnitude. Accounts in the Ethereum world state contain a `codeHash` field, which can be used as a commitment to support verification of contract bytecode properties, such as bytecode size or inclusion of bytecode sub-sequences. With zero-knowledge proof, the L1 one-step proof verifier can verify these properties without the prover revealing the entire contract bytecode.
SNARK-friendly compressed sequencing. We can reduce L2 transaction fees by compressing transaction batches prior to their sequencing. However, this requires either performing decompression on-chain during dispute resolution—which may be prohibitively costly—*or* perform compression or decompression within a SNARK and verify the SNARK on-chain. We plan to leverage (or construct) a SNARK-friendly compression scheme that works well for our use-case.
Data availability layer. We can reduce L2 transaction fees further by posting transaction batches to a separate data availability layer. Recent work [34] aims to make this practical without introducing tenuous trust assumptions, by

partially reusing trust in Ethereum. This is complementary to our work; Specular OSP verifier is modularly designed, enabling us to easily swap out data availability querying (specifically, the verification context) from Ethereum to a different layer, without modifying verification logic.

Decentralized sequencing. Specular's sequencing mechanism should ideally provide both censorship resistance and MEV-resistance (e.g. through commit-reveal and fair ordering). For example, recent work from Kelkar et al. and Zhang et al. introduce schemes that guarantee fair sequencing by committee [35], [36], [37]. However, these schemes also come at the cost of additional coordination overhead and may introduce latency-related vectors for capturing MEV. We plan to investigate approaches that prevent MEV, particularly in the L2 setting.

8. Conclusion

In this work, we argue that ORU architectures that either rely on a custom L2VM or are bound to a specific Ethereum client are difficult to audit and upgrade, and preclude intrinsic client diversity. We introduce an interactive fraud proof system *native* to the EVM to address this problem and describe the properties afforded to an ORU by it. We present an implementation of this approach in Specular.

We demonstrate sufficiently efficient dispute resolution under an EVM-native design, and outline steps to both mitigate costs and reduce trust further in future work.

Acknowledgement

This material is in part based upon work supported by the Center for Responsible, Decentralized Intelligence at Berkeley (Berkeley RDI). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these institutes. We thank Patrick McCorry for his comments that greatly improved the manuscript.

References

[1] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014. https://ethereum.github.io/yellowpaper/paper.pdf.

[2] Patrick McCorry, Chris Buckland, Bennet Yee, and Dawn Song. Sok: Validating bridges as a scaling solution for blockchains. *Cryptology ePrint Archive*, 2021. https://eprint.iacr.org/2021/1589.pdf.

[3] Uriel Feige and Joe Kilian. Making games short. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 506–516, 1997.

[4] Ethereum-optimism/cannon: On chain interactive fault prover for ethereum. https://github.com/ethereum-optimism/cannon. Accessed on 04/26/2022.

[5] OffchainLabs/nitro: Nitro goes vroom and fixes everything. https://github.com/OffchainLabs/nitro. Accessed on 05/30/2022.

[6] Ethereum/go-ethereum: Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum. Accessed on 04/26/2022.

[7] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, volume 1, pages 3–9, 1978.

[8] Eleazar Galano. Infura mainnet outage post-mortem 2020-11-11. https://blog.infura.io/post/infura-mainnet-outage-post-mortem-2020-11-11. Accessed on 05/30/2022.

[9] Go ethereum twitter. https://twitter.com/go_ethereum/status/1431264560019820547. Accessed on 05/30/2022.

[10] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 349–365, 2021.

[11] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018.

[12] Ethereum improvement proposals. https://eips.ethereum.org/. Accessed on 08/06/2022.

[13] Aggelos Kiayias and Philip Lazos. Sok: Blockchain governance. *arXiv preprint arXiv:2201.07188*, 2022. https://arxiv.org/pdf/2201.07188.pdf.

[14] Our pragmatic path to decentralization. https://medium.com/ethereum-optimism/our-pragmatic-path-to-decentralization-cb5805ca43c1. Accessed on 08/06/2022.

[15] Ethereum-optimism/optimism: The optimism monorepo. https://github.com/ethereum-optimism/optimism. Accessed on 04/26/2022.

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

[17] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. Mips: A microprocessor architecture. In *Proceedings of the 15th Annual Workshop on Microprogramming*, MICRO 15, page 17–22. IEEE Press, 1982.

[18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Openethereum/parity-ethereum: The fast, light, and robust client for ethereum-like networks. https://github.com/openethereum/parity-ethereum. Accessed on 04/26/2022.

[20] Nethermindeth/nethermind: Our flagship .net core ethereum client for linux, windows, macos - full and actively developed. https://github.com/NethermindEth/nethermind. Accessed on 04/26/2022.

[21] Introducing evm equivalence. https://medium.com/ethereum-optimism/introducing-evm-equivalence-5c2021deb306. Accessed on 08/06/2022.

[22] Patrick McCorry. Q&A session on plasma, rollups and validating bridges, - cryptocurrency class 2022 – crowdcast, 2022.

[23] Ran Canetti, Ben Riva, and Guy N Rothblum. Practical delegation of computation using multiple servers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 445–454, 2011.

[24] Ran Canetti, Ben Riva, and Guy N Rothblum. Refereed delegation of computation. *Information and Computation*, 226:16–36, 2013.

[25] Bennet Yee, Dawn Song, Patrick McCorry, and Chris Buckland. Shades of finality and layer 2 scaling. *arXiv preprint arXiv:2201.07920*, 2022. https://arxiv.org/pdf/2201.07920.pdf.

[26] OffchainLabs/arbitrum: Powers fast, private, decentralized applications. https://github.com/OffchainLabs/arbitrum. Accessed on 05/30/2022.

[27] Client diversity | ethereum. https://clientdiversity.org/. Accessed on 08/06/2022.

[28] Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. Enter the hydra: Towards principled bug bounties and {Exploit-Resistant} smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1335–1352, 2018.

[29] Grigore Roşu and Traian Şerbănuţă. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79:397–434, 08 2010.

[30] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 313–314. Springer, 2013.

[31] circle-free/merkle-trees: A (hopefully) robust merkle tree library. https://github.com/circle-free/merkle-trees. Accessed on 11/27/2022.

[32] Uniswap/v2-core: Core smart contracts of uniswap v2. https://github.com/Uniswap/v2-core. Accessed on 05/30/2022.

[33] Boltonbailey/formal-snarks-project: A formal verification of the babysnark proof system and others, using the lean theorem prover. https://github.com/BoltonBailey/formal-snarks-project. Accessed on 12/01/2022.

[34] Eigenlayer: A restaking primitive | consensys. https://consensys.net/blog/cryptoeconomic-research/eigenlayer-a-restaking-primitive/. Accessed on 12/06/2022.

[35] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 633–649, 2020.

[36] Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. Order-fair consensus in the permissionless setting. Cryptology ePrint Archive, Report 2021/139, 2021. https://ia.cr/2021/139.

[37] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. Cryptology ePrint Archive, Report 2021/1465, 2021. https://ia.cr/2021/1465.

Appendix A. Dispute Resolution

Dispute resolution protocol. The dispute resolution protocol introduced by Arbitrum draws from the RDoC protocol described in Section 2.3. We describe the protocol as a multi-round recursive bisection of an assertion, but note that it trivially extends to multi-section, which is used in practice to reduce communication.

Suppose, for example, an instance where P_0 and P_1 are two validators that attest to conflicting DAs posted to the L1 bridge. P_1, the challenger, initiates a dispute on the conflicting DA C, against the defending party P_0 that attested to its validity. The L1 bridge serves as referee over a game of synchronous rounds between P_0 and P_1. Note that if a validator does not respond to the bridge during its turn in the allotted time, it automatically forfeits the dispute.

P_1 begins by bisecting the disputed assertion, submitting two commitments to intermediate states between the predecessor of C and C, where the lengths of the state transitions represented by the commitments are equal. Upon receiving the new commitments, P_0 chooses the one it wants to challenge, submitting two new commitments with equal size state transitions. This continues for a logarithmic number of rounds until the disputed computation spans a single instruction. Finally, one party is required to submit a one-step proof to convince the bridge that given the agreed upon state (revealed and verified against a state commitment), executing this instruction results in their asserted final state, which is different from the one committed to previously by the counterparty. The L1 contract simulates the instruction's execution to verify the proof.

If the proof is correct, the opposing party loses; otherwise, the prover loses. Informally, a DA is rejected if and only if all validators that have attested to it by the end of the dispute period have lost disputes, *or* an ancestor to the DA has been rejected. Conversely, a DA is confirmed if and only if validators that have attested to DAs on conflicting branches (if any) have been defeated in disputes, the dispute period has passed *and* its predecessor has been confirmed.

We note that while Arbitrum performs the protocol over gas (i.e. by associating DAs with specific gas consumption quantities), Specular is unable to do so since total gas consumption is not an increasing quantity in Ethereum. Therefore, in order to support EVM opcodes that cost 0 or negative gas (i.e. opcodes that perform refunds), we perform the protocol over steps (instructions and inter-transaction operations) instead, similarly to Nitro and Cannon.

Game theory. Validators are assumed to be rational self-interested parties. They must deposit a stake (i.e. a bond) to the rollup's L1 contract before they are allowed to create or attest to a DA. If a validator attests to a DA, they are agreeing to participate in challenges against it and relinquish their stake if they lose. Specifically, upon losing a dispute, a validator's stake is slashed—a portion going to the dispute winner as a reward (thus also covering their costs), and the rest burned to discourage collusion. The stake size is a security parameter and must be configured to be sufficient to cover at least the cost of dispute resolution.

Appendix B. One-step Proof

B.1. One-step State Definition

We include a summary of how different proof types are composed to construct one-step proofs for each opcode in Table 3. We also elaborate on the structure of the *one-step state* ω and its hash below (including notation defined in [1]):

Block state. An block state $\omega_{\mathbb{b}}$ has following fields to completely describe the state of a finalized block:

blockNumber: The number of the current block.

worldState: $\boldsymbol{\sigma}$ The World State of EVM after finalization of the current block represented in a *trie*.

cumulativeGasUsed: Cumulative gas used by this block and all its ancestors.

blockHashTree: T_b a Merkle tree of 256 previous block hashes (including current block). The block hash of block i is stored at the leaf of index $i\%256$, given block i is the current block or one of the 255 ancestors of the current block.

Opcodes	Proof types							
	stack	memory-R	memory-W	account-R	account-W	storage-R	storage-W	code
MLOAD, KECCAK256, CALLDATALOAD	✓	✓						
MSTORE, MSTORE8	✓		✓					
CALLDATACOPY, RETURNDATACOPY	✓	✓	✓					
BALANCE, SELFBALANCE	✓			✓				
SLOAD	✓			✓		✓		
SSTORE	✓				✓		✓	
RETURN, REVERT	✓	✓			✓			
STOP, INVALID					✓			
CREATE, CREATE2	✓	✓			✓			
SELFDESTRUCT	✓	✓		✓	✓			
CODESIZE, CODECOPY	✓		✓					✓
EXTCODESIZE, EXTCODECOPY	✓		✓	✓				✓
EXTCODEHASH	✓			✓				
CALL, CALLCODE, DELEGATECALL, STATICCALL	✓	✓			✓			
LOG0∼LOG4	✓	✓						
All others (e.g. arithmetic ops)	✓							

TABLE 3: One-step proof composition for each EVM opcode. The required proofs depend on which data structures are accessed by the instruction. In addition to the above proof types, `state`, and `opcode` are included for all opcodes. Some opcodes require multiple proofs of the same type—for example, `RETURN`, `REVERT`, `STOP`, and `INVALID` require an additional `state` proof for the OSS of the last execution step. Some opcodes require special types of proofs not listed in the table, as discussed in Section 4.4.5. Note: "read" and "write" suffixes are abbreviated as `R` and `W` respectively.

The hash of the block state $\omega_{\mathtt{b}}$ is then defined as:

$$H_{\mathtt{OSS}}(\omega_{\mathtt{b}}) = \mathtt{KEC}(\textbf{blockNumber}||r(\textbf{worldState}) \\ ||\textbf{cumulativeGasUsed}||r(\textbf{blockHashTree}))$$

Inter-transaction state. An inter-transaction state $\omega_{\mathtt{int}}$ has following fields to completely describe the state between transactions:

blockNumber: The number of current block where this state is in.

transactionIdx: The idx of the transaction right before this state.

worldState: $\boldsymbol{\sigma}$ The World State of EVM after finalization of the previous transaction represented in a *trie*.

cumulativeGasUsed: Cumulative gas used within the current block.

blockGasUsed: Cumulative gas used before the current block.

blockHashTree: T_b the Merkle tree of 256 previous block hashes (excluding current block).

transactionTrie: T_t the transaction list with all transactions before this state represented in a *trie*.

receiptTrie: T_t the receipt list with receipts of all transactions before this state represented in a *trie*.

The hash of the inter-transaction state $\omega_{\mathtt{int}}$ is then defined as:

$$H_{\mathtt{OSS}}(\omega_{\mathtt{int}}) = \mathtt{KEC}(\textbf{blockNumber}||\textbf{transactionIdx} \\ ||r(\textbf{worldState})||\textbf{cumulativeGasUsed} \\ ||\textbf{blockGasUsed}||r(\textbf{blockHashTree}) \\ ||r(\textbf{transactionTrie})||r(\textbf{receiptTrie}))$$

Intra-transaction state. An intra-transaction state ω has following fields to completely describe the EVM state during its execution:

blockNumber: The number of the block where the current transaction belongs.

transactionIdx: The index of the current transaction in the block.

depth: I_e The execution depth (i.e. how deep the call stack is) in the current point of execution. If the current executing contract is directly called by the transaction, the **depth** is 1.

gas: $\boldsymbol{\mu}_g$ The gas available to the current call.

refund: A_r The gas to refund at the end of execution.

lastDepthState: The OSS of the caller at the time when the current contract is called without calling arguments on stack. If **depth** is 1, the **lastDepthState** is $\boldsymbol{\sigma}_0$, the EVM checkpoint state before the transaction execution [1].

contract: I_a The address of the current executing contract.

caller: I_s The address of the caller.

value: I_v The value that passed along with the call in the current point of execution.

callFlag: The type of calling opcode is used when the current contract is called. 0 for `CALL`, 1 for `CALLCODE`, 2 for `DELEGATECALL`, 3 for `STATICCALL`, 4 for `CREATE`, 5 for `CREATE2`. If **depth** is 1, the **callFlag** is always 0 if the transaction is a contract call, or 4 if the transaction is a contract creation.

out: The starting offset of where the return data should be copied to the caller's memory when the current contract returns.

outSize: The size of the return data that should be copied to the caller's memory when the current contract returns.

pc: $\boldsymbol{\mu}_{\mathtt{pc}}$ The offset of the current executing opcode.

opcode: $I_{\boldsymbol{b}}[\boldsymbol{\mu}_{\mathtt{pc}}]$ The current executing opcode.

codeHash: $\boldsymbol{\sigma}[I_a]_c$ The `codeHash` in the account state of

the current executing contract.

stack: $\boldsymbol{\mu_s}$ The EVM execution stack in the current point of execution, with each element of 256 bits. The hash of the stack is defined of hash chain of the elements in the stack, starting with the bottom of the stack. The hash of an empty stack is zero hash.

memory: $\boldsymbol{\mu_m}$ The EVM execution memory in the current point of execution as a byte array. The byte array is segmented in 256 bits to form cells. The last cell is padded with 0s if its length is less then 256 bits. Then cells are stored in a Merkle tree, the leaf node of which is in format of `offset‖cell`, where `offset` is the offset of the cell.

inputData: $I_{\boldsymbol{d}}$ The input data to the contract being executed. It is built as a Merkle tree similar to **memory**.

returnData: $\boldsymbol{\mu_o}$ The return data of the last returned call, empty if no contract is returned yet. It is built as a Merkle tree similar to **memory**.

committedWorldState: $\boldsymbol{\sigma}$ The World State of EVM in the current point of execution represented in a *trie*.

worldState: $\boldsymbol{\sigma}$ The World State of EVM in the current point of execution represented in a *trie*.

selfDestructSet: A_t the self-destructed set. In Specular, the hash of the self-destructed set is defined as the hash chain of self-destructed contract addresses in order.

logSeries: A_l logs emitted during the transaction. The hash of the log series is defined as the hash chain of all logs emitted in order.

blockHashTree: the same block hash tree as of the inter-transaction before the current transaction.

accessListTrie: (A_a, A_K) the set of accessed account addresses and storage keys for changing gas metering behaviors. In Specular, the access list is constructed as a Merkle Patricia trie where the key is the account address and the value is the storage keys accessed.

The hash of the intra-transaction state ω is then defined as:

$$
\begin{aligned}
H_{\texttt{OSS}}(\omega) = \texttt{KEC}(&\textbf{depth}||\textbf{gas}||\textbf{refund}||H_{\texttt{OSS}}(\textbf{lastDepthState})\\
&||\textbf{contract}||\textbf{codeHash}||\textbf{caller}||\textbf{value}||\textbf{callFlag}\\
&||\textbf{out}||\textbf{outSize}||\textbf{pc}||\textbf{opcode}||size(\textbf{stack})\\
&||H_{\texttt{stack}}(\textbf{stack})||size(\textbf{memory})||r(\textbf{memory})\\
&||size(\textbf{inputData})||r(\textbf{inputData})||size(\textbf{returnData})\\
&||r(\textbf{returnData})||r(\textbf{worldState})||H(\textbf{selfDestructSet})\\
&||r(\textbf{accessList})||H(\textbf{logSeries}))
\end{aligned}
$$

Where $size(\cdot)$ represents the size of the stack/memory/input data/return data in bytes, and $r(\cdot)$ represents the root hash of a Merkle tree or a Merkle Patricia tree.

contract, **caller**, **callFlag**, **value**, **out** and **outSize** are omitted in hash calculation when **depth** is 1, because these parameters are either trivial or can be derived from transaction data sequenced to L1. $r(\textbf{memory})$ is omitted in hash calculation when $size(\textbf{memory}) = 0$; the same goes for $r(\textbf{inputData})$ and $r(\textbf{returnData})$ when $size(\textbf{inputData}) = 0$ or $size(\textbf{returnData}) = 0$.

B.2. One-step State Transition Validity

For a state transition $\omega \rightarrow \omega'$ between two one-step states (either intra-transaction, inter-transaction or block ones), it is considered valid only if (1) it strictly follows the Ethereum specification, and (2) the transition is a single-step (i.e. single opcode execution; transaction initiation or finalization; or, block initiation or finalization). More specifically, the validity of one-step state transitions can be described as follows:

First, the transition between intra-transaction state $\omega_i \rightarrow \omega_{i+1}$ encodes one particular step of EVM opcode execution, the validity of which is determined by the semantics of the opcode to be executed in this step. That is, executing the current opcode of ω_i by EVM execution model exactly results in ω_{i+1}.

Second, the transition between the inter-transaction state and the first intra-transaction state ω_1, $\omega_{\texttt{int}} \rightarrow \omega_1$, encodes the initiation step of the transaction, the validity of which is determined by several conditions, e.g., the validity of the sender's nonce, whether the account has sufficient balance to purchase the required gas.

Third, the transition between the last intra-transaction state ω_n and the inter-transaction state, $\omega_n \rightarrow \omega_{\texttt{int}}$, encodes the finalization step of the transaction, the validity of which is determined by conditions like whether the world state of ω_n and $\omega_{\texttt{int}}$ are equal, whether the gas is correctly refunded, or whether the state is correctly reverted if there is any exception occurred during execution, etc.

Fourth, the state transition of $\omega_{\texttt{b}}^{(B_{i-1})} \rightarrow \omega_{\texttt{int}}^{(B_i)}$ encodes the block initiation step, which is a trivial step and no state is actually changed in this step.

Fifth, when the transaction T is the last transaction in the block B_i, the state transition $\omega_{\texttt{int}}^{(T)} \rightarrow \omega_{\texttt{b}}^{(B_i)}$ encodes the block finalization step, in which the block header is constructed and the block hash is calculated.

For regular transaction T_i, the validity of transition $\omega_{\texttt{int}}^{(T_{i-1})} \rightarrow \omega_{\texttt{int}}^{(T_i)}$ is determined by the following conditions: the change in the world state is valid and the charge in the gas is correct.