Towards Trust-minimized Blockchain Scalability with EVM-native Fraud Proofs

Zhe Ye* *UC Berkeley*
Ujval Misra* *UC Berkeley*
Dawn Song *UC Berkeley*

Abstract

An optimistic rollup (ORU) enables refereed delegation of computation from a blockchain (L1) to an untrusted remote system (L2), by allowing state commitments posted on-chain to be disputed by any party via an *interactive fraud proof* (IFP). This technique has shown promise as a blockchain scalability solution, with existing systems reducing user transaction fees by up to ~20x.

Existing ORUs have focused on adapting existing Ethereum client software to support an interactive fraud proof system, aiming to reuse prior L1 engineering efforts and offer Ethereum Virtual Machine (EVM) semantics at L2. Unfortunately, to do so they bind their on-chain IFP verifier to 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 monocolture 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 the long-term.

In this work, we aim to build a secure, trust-minimized ORU that addresses these problems, while preserving scalability and providing sufficiently efficient dispute resolution. 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 an implementation of this approach, along with an evaluation, in *Specular*, an ORU which leverages an off-the-shelf Ethereum client—modified minimally to support one-step proof generation.

1 Introduction

Public blockchains, such as Ethereum [1], have struggled to scale with rapidly growing demand, resulting in exorbitant transaction fees for users. 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), follows the refereed games model [3], requiring an L2 validator to post and attest to a cryptographic commitment to 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 its defender and submit a proof of an invalid state transition. The commitment is eventually rejected only if the verifier accepts this proof. This mechanism—typically implemented as a smart contract—guarantees the validity of the rollup state, assuming a single honest party exists and follows the protocol.

Recent work by Arbitrum and Optimism, the most popular ORUs deployed today, utilizes an architecture [4, 5] that leverages existing Ethereum client infrastructure, in an attempt to 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]. In order to support IFPs, the Geth EVM Golang source is compiled to a lower-level target instruction set architecture (ISA). The resulting binary is then committed to on-chain, allowing the verifier to enforce its execution at the granularity of the lower-level VM's semantics.

Crucially, this approach does not directly enforce EVM semantics; in fact, it is entirely oblivious to the EVM's existence and semantics. 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

*Equal contribution.

to an unnecessarily large and complex trusted computing base (TCB) that is difficult to independently audit and infeasible to formally verify; and, (3) suffers from a frequently triggered, yet opaque upgrade process—both further increasing 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]. As a result, L2 networks that take this approach are prone to monoculture failures. At L1, the Ethereum network has demonstrated resilience despite occasional mass client failures caused by software bugs [8–10], owing to the diversity of participating clients[1]. Existing L2 approaches feature no comparable safeguards. Consequently, invalid state transitions induced by software bugs can slip by undetected and undisputed—ultimately resulting in a potential 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 (rather than higher-level EVM semantics), the TCB includes the source code of the entire client, custom Golang-to-ISA compiler and binary commitment generator (as illustrated in Figure 1). This leads to a pronounced risk of software vulnerabilities that can impact the security of the ORU, and commensurately increases auditing overheads. Formal verification against an Ethereum specification [11] is also infeasible in this regime, given the complex, computationally unbounded and concurrent nature of these programs.

Third, auditing overheads are compounded by the frequency and lack of transparency in upgrades resulting from a large, complex TCB. Client programs, for example, are upgraded more frequently and less transparently than the underlying formal Ethereum specification itself [6], which undergoes a deliberate proposal process [12]. Upgrades to components of the toolchain specific to the ORU, such as the Golang-to-ISA compiler, are less transparent still.

Moreover, an upgrade to *any* off-chain component requires an on-chain upgrade in-tandem (i.e. to update the binary commitment), 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 the desire to forfeit L1 contract upgrade control in the long-term [14]. However, this remains unrealistic and high-risk, given all off-chain component upgrades—including any vulnerability patches—must be reflected in the on-chain 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 community. 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 and transparent (compared to Ethereum as a baseline), 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 and most popular public programmable blockchain network; 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 diversity in software versions.

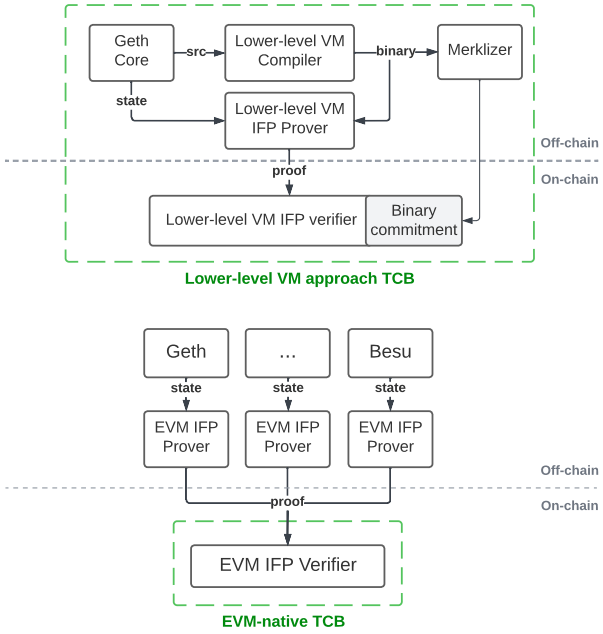


Figure 1: TCB comparison. The top diagram illustrates the complexity and size of the TCB of the lower-level VM approach. The bottom diagram shows how the TCB of an EVM-native ORU is minimal in size.

EVM-native IFP We therefore argue that the verification of semantics beyond those formally defined by the Ethereum specification is counterproductive, due to the disadvantages outlined. We instead propose an *EVM-native* interactive fraud proof approach, which directly conducts verification of EVM semantics. That is, we propose an efficiently verifiable one-step proof construction *directly* over the state transition resulting from the execution of a *single EVM instruction* (or inter-transaction operation). As such, the proof should be feasible to construct by any existing Ethereum client. This differs from Arbitrum, which targets a custom instruction set architecture, the AVM [16], and ongoing open-source efforts which target custom versions of lower-level ISAs—MIPS [17] by

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: Comparison of existing optimistic rollup systems and Specular. We include concurrent efforts in the Optimism and Arbitrum communities for completeness, but note the ongoing status of their work. In particular, to our knowledge Cannon does not yet have a fully specified and working challenge protocol. The maintainers of Cannon have also proposed an approach to enabling a limited form of client diversity, which we discuss in Section 7.1.

Optimism Cannon [4] and WebAssembly [18] by Arbitrum Nitro [5])—compiled down to from an Ethereum client (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 fraud proof 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 an EVM-native fraud proof 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 [4, 5], which suffer from client lock-in.

Next, an *EVM-native* fraud proof system reduces the scope of the TCB to *only* the EVM IFP verifier on L1. The TCB is therefore minimal in size, practically resulting in a more secure and auditable system. Additionally, this results in fewer and more transparent upgrades; specifically, only upgrades to the formal Ethereum specification (rather than to a particular client program) can trigger a TCB upgrade. Upgrades to client programs can be made independently of the TCB, reducing governance overhead and making forfeiture L1 contract ownership more plausible.

Finally, an *interactive* fraud proof system with an efficiently verifiable one-step proof allows for efficient dispute resolution, as shown by prior work.

Contributions The key challenge in our approach is to support one-step proof generation directly over all EVM instructions and inter-transaction operations. 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 straightforward 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. This enables us to adapt the EVM to L2 with negligible modifications.

We introduce *Specular*, an end-to-end implementation of an ORU with an EVM-native L2VM, called the *SpecularVM*. Specular supports an efficient interactive fraud proof protocol, while avoiding preclusion of multiple L2 clients and remaining simple to maintain. Figure 2 illustrates the high-level components of Specular's L2 client. In summary, we make the following key contributions:

- We motivate the necessity of an *EVM-native* interactive fraud proof approach that only verifies EVM semantics and nothing more, enabling 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 (Go Ethereum, or Geth [6]) to the interactive fraud proof setting. Our implementation modifies only 100 lines-of-code within Geth itself, demonstrating the simplicity of the architecture and feasibility of extending to multiple L2 clients. Specifically, only 20 lines-of-code are added to the Geth core to support one-step proof generation.
- We integrate these components to implement Specular, the first optimistic rollup to achieve (1) a minimal TCB which improves security, auditability and upgradeability, (2) support for permissionless participation of multiple Ethereum clients, 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

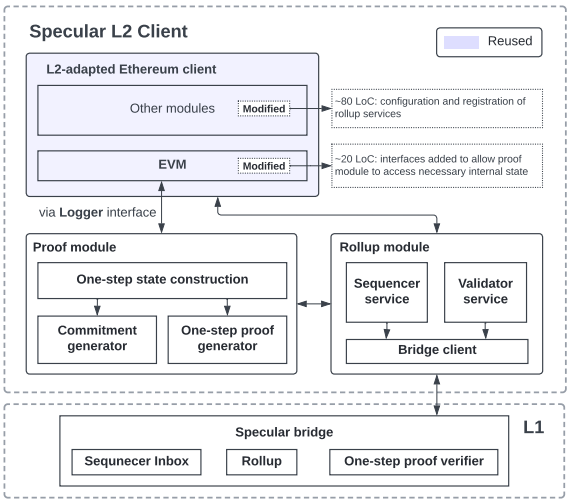


Figure 2: Specular L2 client overview. The L2 client is composed of an L2-adapted Ethereum client, such as Geth (with an EVM implementation modified to support proof generation); a proof module that generates one-step proofs; and a rollup module that is responsible for sequencing and/or validation.

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 within 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 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 *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 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 $I := (I_a, I_o, I_p, \boldsymbol{I_d}, I_s, I_v, \boldsymbol{I_b}, I_H, I_e, I_w)$ is a tuple of read-only data 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 $\boldsymbol{I_d}$, invoker account address I_s, monetary value I_v, bytecode $\boldsymbol{I_b}$, 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 $A := (A_s, \boldsymbol{A_l}, \boldsymbol{A_t}, A_r, \boldsymbol{A_a}, \boldsymbol{A_K})$, containing the self-destruct set A_s, the series of logs emitted $\boldsymbol{A_l}$, the set of touched accounts $\boldsymbol{A_t}$, the refund balance A_r, the set of accessed account addresses $\boldsymbol{A_a}$ and the set of accessed storage keys $\boldsymbol{A_k}$.

The 4-tuple $(\boldsymbol{\sigma}, \boldsymbol{\mu}, I, A)$ 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 (and live).

In this approach, if the servers unanimously agree on a result, the client accepts it immediately. If two 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. On identifying the inconsistency at the single instruction-level, the client determines which party is dishonest by locally evaluating the instruction, and accepts the result claimed by the honest server. This protocol extends trivially to an n-server disagreement.

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

[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.

Type	Properties							
	Ecosystem compatibility	L1 client reuse	Fraud proof target	Level of equivalence	TCB	Upgrade transparency	On-chain upgrade frequency	Intrinsic client diversity
EVM-compatible	Limited	✗	Custom VM	Solidity	Large	✗	Frequent	✗
EVM-equivalent	Full	✓	Custom ISA	Client binary	Large	✗	Frequent	✗
EVM-native	Full	✓	EVM	Ethereum spec	Minimal	✓	Rare	✓

Table 2: Comparison of types of EVM reuse. Each degree of EVM reuse gives rise to unique properties in the ORU. Ecosystem compatibility refers to both Ethereum smart contracts and the Ethereum toolchain broadly. Upgrade transparency and frequency refer to the transparency and frequency of upgrading the TCB, including the L1 verifier.

Therefore 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.

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-bounded client and L2 parties the more powerful servers. An ORU is secured by a *bridge* residing on L1, and is typically implemented as a smart contract (or contracts) [2].

ORUs decouple transaction sequencing from state computation; a sequencer decides on an ordering off-chain and posts the ordered transactions to L1, while validators (fulfilling the RDoC server role) read and execute these transactions in an off-chain virtual machine (L2VM). 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 a state commitment (also referred to as a *disputable assertion*, or DA) and when necessary, initiate a dispute on L1 within the agreed upon *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 gas of an L2 block, with existing protocols (including Arbitrum) allowing for as much as a week [25]. We describe the full dispute resolution protocol used by Arbitrum [16], which we also utilize in this work, in Appendix A.

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 and follows the protocol, an invalid state will not be confirmed on-chain.

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 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 a lower-level target ISA (MIPS by Cannon, WASM by Nitro). The resulting binary is then committed to on-chain, allowing the verifier to enforce its execution at the granularity of the lower-level VM's instruction semantics. Crucially, this lower-level VM approach does not directly enforce EVM semantics; in fact, it is oblivious to them. Instead, it enforces the execution of a *specific* client binary. In the next section, we enumerate over the challenges that manifest due to this design and how an EVM-native IFP resolves them.

3 Our Approach: EVM-native IFP

To solve 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.

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 to prevent all consensus failures), no current L2 solution supports any form of client diversity. 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 overall if they manage to win a threshold of the challenge phases invoked. 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, participation of a client program is *permissioned*. That is, a client program's compiled binary must be committed to on-chain as a whitelisted program for it to be allowed to participate 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 participate and be 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 [26]. 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 On the other hand, 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. Additionally, validators are afforded a choice in running their client program of choice, providing simpler devops.

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 formally verifiable to ensure its trustworthiness. Existing ORUs fall 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 equivalence of the enforced semantics to that of the EVM. The enforced semantics of the ORU VM are determined implicitly 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. Moreover, with extrinsic client diversity, this problem is exacerbated further; semantics are determined by *majority consensus* (manifesting through the dispute mechanism)—requiring audits for *all*, or at least a majority of, client programs and compilers.

The TCB therefore includes not only the verifier, but also every client program, along with the compiler toolchain and binary commitment generator associated with each program. The bloated size and complexity of the TCB creates a pronounced risk of vulnerabilities that can impact the security of the ORU, and commensurately increases auditing overheads.

We emphasize that it isn't possible to piggyback off of security audits conducted by the original client program maintainers because the program must be custom-tailored to support one-step proof generation. Furthermore, 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 binary code 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. For this reason, the TCB of an EVM-native ORU includes only the on-chain verifier. The limited scope allows it to be both more easily auditable and entirely formally verifiable with respect to a formal EVM specification, as we touch on below.

Formal verifiablity Formal verification—and in particular, KEVM [11]—provides a straightforward means to verify the on-chain verifier. KEVM is an executable formal specification of the EVM implemented using the K-Framework [27] that supports the formal verification of Solidity smart contracts. KEVM is therefore perfectly complementary to Specular because it provides both a formal specification and a suite of verification tools, enabling us to formally verify our on-chain verifier. 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's 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 doesn't provide any assurances regarding the enforcement of EVM semantics. To achieve such assurances, Geth itself—along with its compilation toolchain—would have to be formally verified; however, given the computationally unbounded and concurrent nature of these programs, this is effectively impossible. An EVM-native ORU's TCB is therefore formally verifiable, while that of Optimism/Arbitrum is not.

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 can stabilize in tandem with Ethereum. This is because Specular's developers could forfeit L1 contract upgrade control without forfeiting the ability to ship vulnerability patches, bug-fixes and performance improvements to clients. This is not possible with the lower-level VM approach, due to the inherent link between the client program 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 proposal (EIP) process. 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 aims to convince the verifier that given an initial L2VM state, executing the current instruction will result in a transition to the asserted final L2VM state. The system must address the following requirements.

1. **EVM-native one-step proof.** The one-step proof attests to the validity of a state transition at the granularity of a single EVM instruction. All EVM instructions are supported.

2. **Efficient.** The one-step proof generation procedure runs quasilinearly in the size of L2VM state b; the size of the proof is logarithmic in b, linear in contract size and linear in bytes accessed. Concretely, a one-step proof should fit within a single on-chain transaction (on Ethereum, this is 30 million gas).

3. **Extensible.** The one-step proof can be constructed exclusively from state specified by and accessible from the EVM, enabling reuse of any of numerous existing clients.

The key challenge is to design a one-step proof for the EVM that is sufficiently efficient to verify on-chain. This requires commitment schemes for the stack, memory and storage that support efficient partial reveals and updates.

4.2 Definitions

Our L2VM, which we refer to as *SVM* inherits all state from the EVM. We therefore borrow formalism used to describe the EVM (see Section 2) wherever convenient.

The *one-step state* (OSS) is the state between EVM transaction execution steps (i.e. EVM instruction execution, transaction initiation or finalization). It has two forms: *intra-transaction state* and *inter-transaction state*.

Intra-transaction state The *intra-transaction state* ω of the SVM 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 OSS and the commitment to it $H_{\texttt{OSS}}$ is included in Appendix B). `KEC` denotes the Keccak-256 cryptographic hash function [28]. 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 that is consistent with the corresponding commitment field in the state proof, to prove the validity of the state transition in the component. Since opcodes do not use every component (for example `ADD` does not access or modify the memory or the world state), we provide a modular design of the one-step proof, 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. Formally, it is defined as $(\boldsymbol{\sigma}, g)$. The commitment to it differs from the intra-transaction OSS, and is defined as $H_{\texttt{OSS}}((\boldsymbol{\sigma}, g)) := \texttt{KEC}(r(\boldsymbol{\sigma})||g)$, where $r(\boldsymbol{\sigma})$ is the root of Merkle Patricia tree that stores $\boldsymbol{\sigma}$.

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

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 $\omega_{\texttt{int}}^{(T_{i-1})} \to \omega_{\texttt{int}}^{(T_i)}$.

For a state transition $\omega \to \omega'$ of two one-step states (either intra-transaction or inter-transaction ones), it is considered valid if in precisely one step of EVM single instruction execution, transaction boostrapping or finalization, state ω transits to state ω'.

First, the transition between intra-transaction state $\omega_i \to \omega_{i+1}$ 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, $\omega_{\texttt{int}} \to \omega_1$, encodes the initiation step of the transaction, the validity of which is determined by several conditions, e.g., the validity of the transaction signature and 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, $\omega_n \to \omega_{\texttt{int}}$, encodes the finalization step of the transaction, the validity of which is determined conditions like whether the world state of ω_n and $\omega_{\texttt{int}}$ are equal, whether the gas is correctly refunded, etc.

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

Note that if during any transition $\omega_{\texttt{int}} \to \cdots \to \omega \to \omega'_{\texttt{int}}$, the validity constraints of transition are violated (e.g. insufficient balance to transfer), an EVM exception occurs and the execution of the current contract creation or message-call is halted and reverted. In this case, the valid state transition should be ω to some state that reflects the exception. If the execution depth of ω is 1, the valid transition instead should be $\omega \to \omega''_{\texttt{int}}$ (i.e. termination of execution), where the world state of $\omega''_{\texttt{int}}$ should be identical to $\omega_{\texttt{int}}$ so changes in the world state are reverted; however the gas charge and emitted logs are preserved. Otherwise, the present message-call or contract creation should immediately revert and return to the last execution, which is similar to executing the `REVERT` opcode on ω instead of the current instruction of ω.

4.4 Proof System

Suppose that upon executing the current instruction $I_{\boldsymbol{b}}[\boldsymbol{\mu}_{\texttt{pc}}]$, 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 one-step proof must convince a verifier that the transition $\omega \to \omega'$ (equivalently, the transition from $h \to h'$) is valid.

More formally, the one-step proof system is a tuple (ω, ω', P, V), where P is the one-step proof generator, and V is the one-step proof verifier. P is defined as $\pi := P(\omega, \omega', I)$. π is a one-step proof for 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 as the sender and recipient, or block contexts associated with transactions like timestamp and block number.

A one-step proof π is a composition of proofs, each of which proves the validity of a part of the transition $\omega \to$

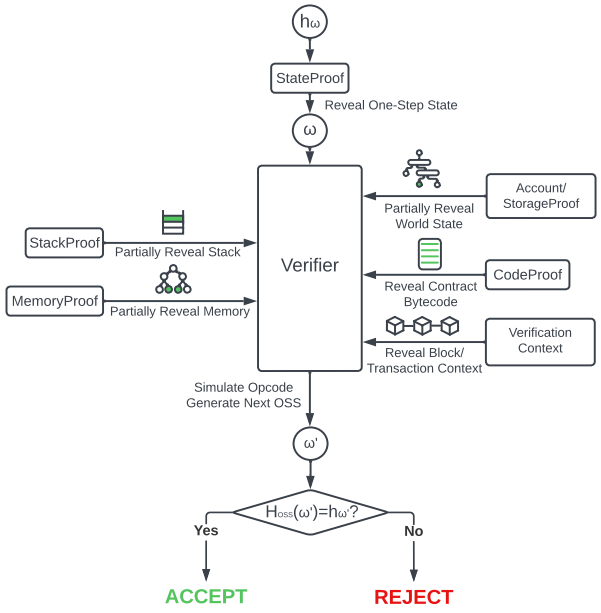


Figure 3: One-step proof verification. An illustration of the flow of inputs and outputs within the one-step proof verifier.

ω'. This proof establishes that (1) ω is consistent with h, (2) executing $I_{\boldsymbol{b}}[\boldsymbol{\mu}_{\text{pc}}]$ while in state ω yields ω', and (3) ω' is consistent with h'.

The types of proofs that π consists of depend on the opcode of the current instruction and consequently which SVM data structures are read from or written to (see Table 3 for a summary). Specifically, each type of proof corresponds to the authenticity and integrity of the transition of the OSS itself, or that of a component of the OSS (i.e. that of the stack, memory, or world state). There are also some special types of proof that handles the execution exception or the transitions involving the inter-transaction state. We introduce each type of proof below.

4.4.1 Preliminary proofs

We provide the following proofs for *all* instructions.

First, a *state* proof establishes the authenticity 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 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. 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 integrity of the next instruction to be executed (at offset $\boldsymbol{\mu}'_{\text{pc}}$). 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 zero-knowledge proof techniques to further reduce the proof size and briefly comment on this in Section 7.2.

Third, an *exception* proof attests to certainty of the execution exceptions of EVM, which is provided by the prover when an execution exception (e.g. ran out of gas) occurred, instead of the normal proof based on opcodes. It includes a error code field to indicate which particular exception occurred, along with different types of proofs and additional information according to the exception type. For example, the ran-out-of-gas exception only requires the preliminary proofs to confirm the available gas to the current call is indeed insufficient, while the insufficient balance exception or contract address collision exception may require an *account-read* proof.

4.4.2 Stack

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

$$H_{\texttt{stack}}(\boldsymbol{\mu}_{\boldsymbol{s}}) = \begin{cases} \texttt{KEC}(()) & \text{if } n = 0 \\ \texttt{KEC}\left(H_{\texttt{stack}}\left(\boldsymbol{\mu}_{\boldsymbol{s}}[0, n-1]\right) || \boldsymbol{\mu}_{\boldsymbol{s}}[n-1]\right) & \text{if } n > 0 \end{cases}$$

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

Then, the stack proof is the tuple (h, h_0, p), where h_0 is the intermediate stack hash, $p := \boldsymbol{\mu}_{\boldsymbol{s}}[|\boldsymbol{\mu}_{\boldsymbol{s}}| - \delta, |\boldsymbol{\mu}_{\boldsymbol{s}}|]$, is the subsequence of the δ top elements popped, where $|\boldsymbol{\mu}_{\boldsymbol{s}}|$ is the size of the stack; and $q := \boldsymbol{\mu}'_{\boldsymbol{s}}[|\boldsymbol{\mu}'_{\boldsymbol{s}}| - \alpha, |\boldsymbol{\mu}'_{\boldsymbol{s}}|]$ is the subsequence of α elements pushed.

Given the proof, the verifier computes $h_k^{\texttt{POP}} := \texttt{KEC}\left(h_{k-1}^{\texttt{POP}} || p[k]\right) \quad \forall \ 0 < k \le \delta$, where $h_0^{\texttt{POP}} = h_0$ and $p[i]$ is the i-th element of p, and checks $h = h_\delta^{\texttt{POP}}$ to verify

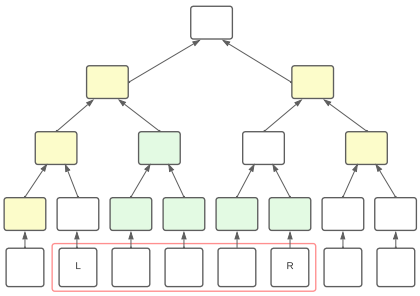


Figure 4: Merkle range proof. The elements accessed are boxed in red. Green nodes represent hashes which can be computed instead of being provided by the prover (i.e. if we were to avoid providing Merkle proofs for *all* elements). Yellow nodes represent hashes that must still be provided.

the authenticity of the popped elements. The verifier then simulates the execution of $I_b[\boldsymbol{\mu}_{\text{pc}}]$ to compute the sequence q of α elements to push onto the stack. Finally, it computes $h_k^{\texttt{PUSH}} := \texttt{KEC}(h_{k-1}^{\texttt{PUSH}} || q[k]) \ \forall \ 0 < k \leq \alpha$, where $h_0^{\texttt{PUSH}} = h_0$ and $q[i]$ is the i-th element of q, to get $h_\alpha^{\texttt{PUSH}}$, 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 SVM memory space. $H_{\texttt{OSS}}$ commits to a Merkle tree root computed over the entire contents of the SVM memory space $\boldsymbol{\mu_m}$, where each leaf i is $\texttt{KEC}(\boldsymbol{\mu_m}[32i, 32i+32])$.

The *memory-read* proof (i, c, p, r) attests to the authenticity of the byte array c starting at offset i of $\boldsymbol{\mu_m}$, with respect to memory Merkle root r. A naive approach would require p, the proof of authenticity of c, to contain Merkle proofs for each of its element, resulting in a proof size in $O(|c| \log |\boldsymbol{\mu_m}|)$. However, we can reduce the proof size by a factor of $|c|$ by eliminating redundancy in the provided hashes. We therefore provide a single range proof across all elements; p is the Merkle multi-proof for the left-most and right-most elements. This information is sufficient for the verifier to reconstruct the Merkle root, as illustrated in Figure 4. Merkle proofs for each interior element are not required, since their neighboring elements are revealed.

Likewise, the *memory-write* proof attests to the integrity of the entire memory after a write. That is, it proves the authenticity 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 the authenticity of c in r. It then verifies (i, c', p, r'), where r' is the new memory Merkle root to verify. This proves that only the memory starting at location i was modified.

The memory-read proof and memory-write proof are also applicable to the calldata $\boldsymbol{I_d}$ 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 authenticity of the account state $\boldsymbol{\sigma}[a]$ associated with address a, as read by an instruction. The proof provides information corresponding to each account field (`nonce`, `balance`, `storageRoot`, `codeHash`) sufficient to compute the account hash. The verifier computes the hash of the concatenated fields, and with the MPT proof, computes and verifies the root.

Likewise, the *storage-read* proof attests to the authenticity of a value read from a storage slot in the MPT with root $\boldsymbol{\sigma}[a]_s$ (assuming the authenticity 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 authenticity 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 authenticity 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 The *account-write* proof attests to the integrity of the Ethereum world state after an account's fields are written to. In addition to account-read proof, it provides the modified account object field values, as well as the MPT proof of the new account object. The *storage-write* proof attests to the integrity of an account object after a slot in its storage is written to. This includes a storage-read proof for the content during initial state $\boldsymbol{\sigma}$, the new content to be written to the storage slot, and its account MPT proof. A storage-write proof is accompanied by an account-write proof, since the account object is modified on any storage write.

4.4.5 Inter-transaction

The *inter-transaction* proofs attest to the validity of the transaction initiation and finalization steps, as discussed in Section 4.3. Depending on the type of transaction, the inter-transaction proof may include different types of proofs and additional information. For example, the *pre-transaction* proof always includes an *account-read* proof of the sender and the contract to be executed if it is not a regular transaction.

Similarly, an *account-write* proof is required for the *post-transaction* proof to attest the refund of the remaining gas and also code changes if the transaction is a contract creation.

4.4.6 Verification

The one-step proof verifier verifies the authenticity of the initial OSS ω, simulates the execution of the current SVM instruction—given sufficient context by the prover to do so correctly—and depending on the computed hash of the resultant final OSS $H_{\text{OSS}}(\omega')$, 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, verify the validity of the state transition by inter-transaction proof (defined in Section 4.4.5) 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`;
4. If the execution exception proof (defined in Section 4.4.1) indicates that there is an execution exception, verify that exception and `ACCEPT` if that exception indeed occurred; otherwise, `REJECT`;
5. If the execution exception proof indicates that there is no execution exceptions, 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`;
6. Simulate the execution of the $I_{\boldsymbol{b}}[\boldsymbol{\mu}_{\text{pc}}]$, given C and π. C and π partially reveal the EVM execution environment to the simulator, which executes $I_{\boldsymbol{b}}[\boldsymbol{\mu}_{\text{pc}}]$ 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`;
7. 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 $\omega \to \omega'$, 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. The worst-case space complexity of the one-step proof is $O(\log s_\sigma + \log s_m + \log s_i + \log s_o + s_c + n_m + n_i + n_o + n_c)$. The worst-case time complexity of one-step proof verification (or gas cost equivalently) is $O(\log s_\sigma + n_m \log s_m + n_i \log s_i + n_o \log s_o + n_c + s_c)$.

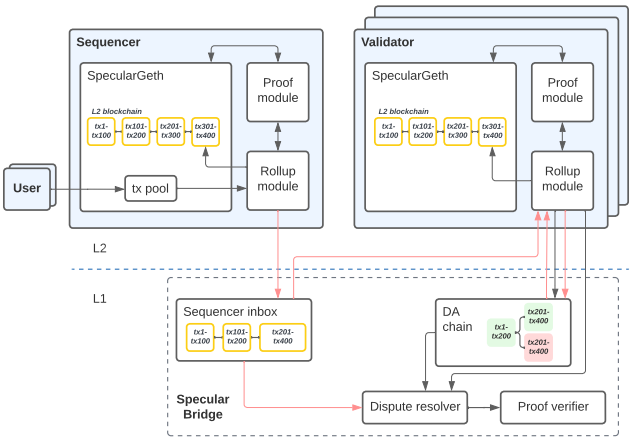


Figure 5: Specular architecture. The sequencer and validator node implementations are simple wrappers around (modified) Geth. Gray arrows represent control flow and pink arrows represent data flow. A sequencer posts data to the sequencer inbox. Validators read this data, and post, attest to or dispute DAs on the DA chain contract.

5 Implementation

We implement the proposed new optimistic rollup system, Specular. Specular comprises of a fraud proof system (outlined in Section 4), an L2 sequencer, 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 Go-Ethereum (Geth) fork [6] (which we refer to as SpecGeth), as illustrated in Figure 5 [15]. Note that SpecGeth in Figure 5 corresponds to the L2-adapted Ethereum client component in Figure 2. The concrete difference between SpecGeth and Geth is only 100 lines-of-code, including 20 additional lines in Geth core that expose select internal state to the proof module, and 80 lines changed outside Geth core for rollup service configuration. 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 [29]—the key differences being in the structure of the disputable assertion and one-step proof verifier. The bridge functionality is split across four 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 initiation of a dispute to referee multisection; and (4) the one-step proof verifier, which is invoked by the dispute resolver to verify a one-step proof.

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 SVM through the `EVMLogger` API, which records sufficient SVM 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 ~3400 lines of Go code, including 520 lines for OSS definitions and construction, 510 lines for one-step proof definitions and encoding, and 2400 lines for one-step proof generation.

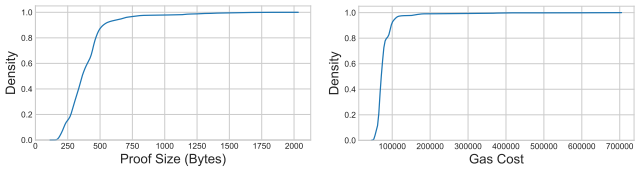
The one-step proof verifier in the L1 bridge comprises of two key contracts: `OneStepProof` and `ProofVerifier`. The `OneStepProof` contract implements the definition and decoding of one-step proof in ~400 lines of Solidity code. The `ProofVerifier` contract implements the one-step proof verifier illustrated in Figure 3 for each type of one-step proofs, and an SVM opcode simulator to generate the commitment for the next one-step state. The `ProofVerifier` is implemented in ~2110 lines of Solidity code.

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 timestamp), to the inbox contract on L1. This data is stored by the blockchain 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. The bridge client—running concurrently on the same machine—periodically gathers transactions from the SpecGeth instance's local blockchain, and constructs batches to post to L1.

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.



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

Figure 6: 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.

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). The connection to the L1 blockchain is handled by another, unmodified, Geth instance.

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 by the contract automatically.

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.

6 Evaluation

In this section, we provide some key performance measurements for Specular. We argue that our system is practical in real world conditions and supports sufficiently efficient dispute resolution. 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.

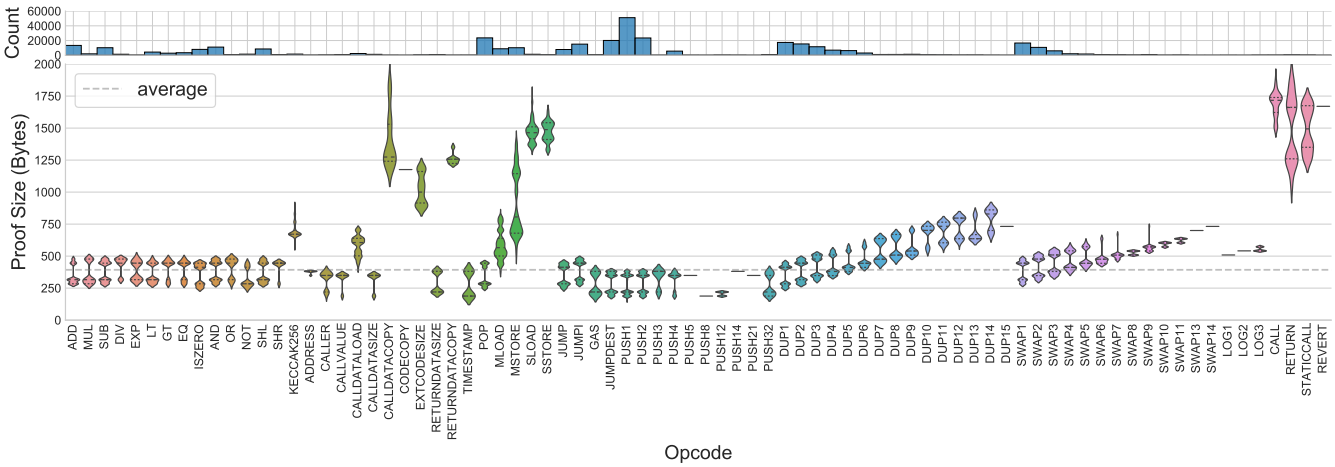


Figure 7: Proof sizes for Uniswap v2 transactions (sans contract effects). Illustrates proof size distribution on a per-opcode basis, along with the distribution of opcodes in the contract.

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 [30], 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 353,594 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 6 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 for verification of authenticity of the bytecode. We plan to replace this approach with a zero-knowledge proof to mitigate the cost of including the bytecode. Therefore, to better illustrate the actual properties of the one-step proof evaluated, Figures 6, 7 and 8 do not include the effects of contract bytecode (however, the associated text in this Section does).

One-step proof size We measure the latency of generating a one-step proof (across ~300,000 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 390B (min. 160B, max. 1990B). The proof sizes *with* contract bytecode included for opcode and code proofs are an average of 12.6KB (min. 2.4KB and max. 19.4KB). This size depends largely on the contract in which the step occurs. Figure 7 provides a distribution of proof sizes on a per-opcode basis. As a 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 Arbitrum and Optimism v2 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 argue that the sizes of OSPs generated by Specular are acceptable in a real world setting. This is also shown by the cost of verification, which includes the submission of the one-step proof to L1 as the `calldata` to verification contract. We demonstrate in the next section 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 78,000 gas (min. 53,100, max. 696,500). 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 gas limit. Figure 8 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 462,600 gas (min. 125,500, max. 1,146,000). The increases

of gas costs include the cost for including contract bytecode in calldata, copying it into memory and hashing to verify its authenticity. 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 1,500,000 gas (i.e. only 5% of the Ethereum block gas limit). Thus, the verification cost of our approach in the worst case is still acceptable 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 8. `SSTORE` requires four MPT proofs, including two for the account and storage (before and after 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 for Merkle range proof verification).

7 Discussion

7.1 Related Work

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 EVM implementation while supporting interactive fraud proofs [4, 5]. In their approaches, the EVM implementation is compiled down to a lower-level instruction set architecture (WebAssembly [18] and MIPS [17] respectively); one-step proofs are generated and verified over instructions in the reduced ISA, instead of those of the EVM.

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

Moreover, the low-level ISA execution traces of different client types likewise differ, presenting an overwhelming challenge to achieving client diversity at L2. One proposal to do so requires dispute participants to run dispute resolution for each L2 client [14]. Not only is this less efficient and more expensive (linear in the number of clients), it also increases maintenance overhead by expanding the size of the TCB further, and requires governance decisions on which clients to support. That is, this approach is *permissioned*. We argue that ours is comparatively simpler, more efficient and permissionless; since it is EVM-native, any Ethereum client can be used without permission. Our design therefore creates more favorable circumstances for multiple L2 clients to emerge.

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

7.2 Future Work

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.

Decentralized sequencing The sequencing mechanism should provide both censorship resistance and fair ordering (i.e. MEV-resistance). Recent work from Kelkar et al. and Zhang et al. introduce schemes that guarantee fair sequencing by committee, at the cost of additional coordination overhead [31–34]. These approaches are also compatible with our work.

SNARK-friendly compressed sequencing We can reduce L2 transaction fees by compressing transaction batches prior to their sequencing. However, this requires the ability to either perform decompression on-chain during dispute resolution—which may be prohibitively costly—*or* perform compression or decompression within a zk-SNARK and verify the SNARK on-chain. We are interested in leveraging (or constructing) a SNARK-friendly compression scheme that works well for our use-case.

Formal verification We plan to formally verify our L1 verifier against an existing Ethereum specification, such as that of KEVM [11] (built on K-framework [27]) in order to improve the trustworthiness of Specular's TCB and make it practical to forfeit L1 contract upgrade control.

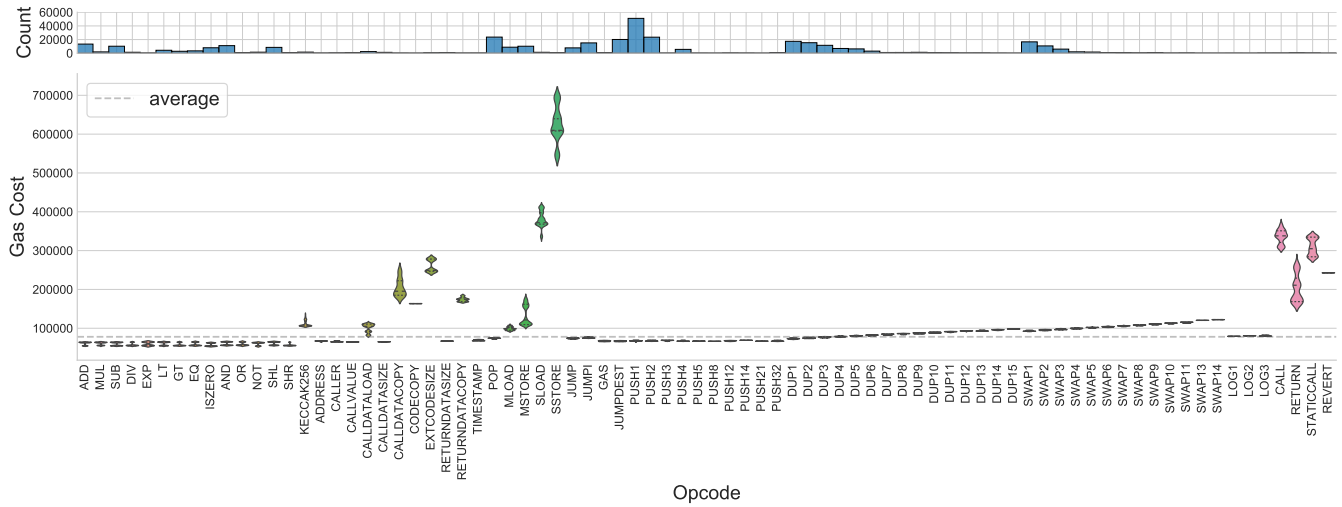


Figure 8: Verification gas costs for Uniswap v2 transactions (sans contract effects). Illustrates gas cost distribution on a per-opcode basis, along with the distribution of opcodes in the contract.

Extension to non-Ethereum blockchains Since Specular is EVM-native, it can be easily generalized to support any EVM-compatible L1 blockchain. It is also possible to introduce the benefits of an EVM-native system—including EVM support and L2 client diversity—to EVM-incompatible blockchains without any further modifications to Specular at L2, as long as the L1 system provides a sufficiently expressive VM for EVM one-step proof verification. Algorand [35] is one such example. We plan to extend Specular to scale EVM-incompatible blockchains in the future.

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 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.

References

[1] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

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

[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.

[14] Our pragmatic path to decentralization. https://medium.com/ethereum-optimism/our-pragmatic-path-to-decentralization-cb5805ca43e1. 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.

[26] 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.

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

[28] 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.

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

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

[31] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Annual International Cryptology Conference*, pages 451–480. Springer, 2020.

[32] 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.

[33] 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.

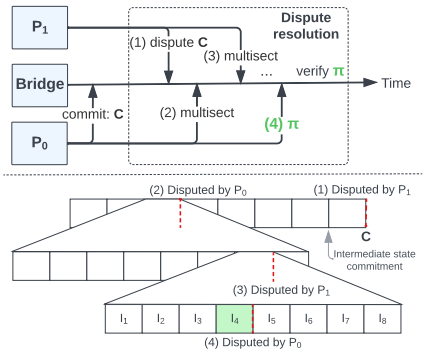


Figure 9: Dispute resolution protocol. An example of the execution of the dispute resolution protocol with an eight-way multisection each round. P_0 commits to a new state and P_1 subsequently initiates a dispute against the new commitment. π is a one-step proof of the transition resulting from executing instruction I_4.

[34] 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.

[35] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

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, the instance illustrated in Figure 9, 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 (by default, upon creation of the DA). 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 (which is guaranteed to exist), executing this instruction results in a final state 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 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.

B One-step Proof

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]):

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: μ_g The gas available to the current call.

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

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	✓	✓			✓			
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`, `opcode` and `exception` proofs 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. Note: "read" and "write" suffixes are abbreviated as `R` and `W` respectively.

lastDepthState: The OSS of the caller at the time when the current contract is called without calling arguments on stack. If **depth** is 0, the **lastDepthState** is σ_0, the EVM World State before the transaction execution.

contract: I_a The address of the current executing contract.

codeHash: $\sigma[I_a]_c$ The `codeHash` in the account state 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 0, 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: μ_{pc} The offset of the current executing opcode.

opcode: $I_b[\mu_{\text{pc}}]$ The current executing opcode.

stack: μ_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: μ_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_d The input data to the contract being executed. It is built as a Merkle tree similar to **memory**.

returnData: μ_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**.

worldState: σ The World State of EVM in the current point of execution represented in a *trie*.

The hash of the one-step state ω is then defined as:

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

Where $\mathit{size}(\cdot)$ 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.

$h(\textbf{lastDepthState})$, **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 $\mathit{size}(\textbf{memory}) = 0$; the same goes for $r(\textbf{inputData})$ and $r(\textbf{returnData})$ when $\mathit{size}(\textbf{inputData}) = 0$ or $\mathit{size}(\textbf{returnData}) = 0$.