Optimism

Security Assessment

September 23, 2022

Prepared for:
Matthew Slipper
Optimism

Prepared by: **Michael Colburn, David Pokora**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Optimism engaged Trail of Bits to review the testing strategy of its Optimistic rollup engine, Optimistic L2 `go-ethereum` fork, and Bedrock smart contracts. From August 22 to September 23, 2022, a team of two consultants conducted a review of the client-provided source code, with eight person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system, including access to the source code and documentation. We performed automated analysis against the project targets, as mentioned in the Automated Testing section of the report.

Summary of Findings

The audit uncovered a significant flaw that could impact system confidentiality, integrity, or availability. A summary of the findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-OPTEST-1**
  The `GasPriceOracle` contract deployed to L2, which is used to update L1 costs charged in L2, can be misconfigured to set gas to a price that does not allow any transactions to be processed. This may even block future attempts to reset the `GasPriceOracle`.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Cara Pearson, Project Manager
cara.pearson@trailofbits.com

The following engineers were associated with this project:

Michael Colburn, Consultant
michael.colburn@trailofbits.com

David Pokora, Consultant
david.pokora@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
August 18, 2022	Pre-project kickoff call
August 29, 2022	Status update meeting #1
September 6, 2022	Status update meeting #2
September 19, 2022	Status update meeting #2
September 26, 2022	Delivery of report draft
September 26, 2022	Report readout meeting

Project Goals

The engagement was scoped to provide a security assessment of the Optimism team's `op-geth`, `op-node`, and bedrock smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- What invariants should be tested across all the project targets?
- Are there any gaps in existing testing methodology?
- Can any existing unit tests be better served with an accompanying fuzz test?
- How could the slither API be used to statically analyze smart contracts within Optimism?
- Are there any recommendations which can be made with respect to improving testability of some targets?
- Generally, does the system behave as expected when tested under various conditions?
    - Are blocks produced in a timely fashion?
    - Are access controls in place to prevent someone from submitting deposit transactions over L2 RPC?
    - Does the system work end-to-end? Do individual components of `op-geth` and `op-node` behave as expected?
    - Are data structures serialized and deserialized without data loss?
    - Are balances and fees charged as expected?
    - Does the system behave as expected when forks are encountered?

Project Targets

The engagement involved a review and testing of the targets listed below.

Optimism (op-node, op-e2e, bedrock contracts)

Repository	https://github.com/ethereum-optimism/optimism
Version	`b31d35b67755479645dd150e7cc8c6710f0b4a56`
Types	Golang, Solidity
Platforms	Linux, macOS, Windows, Ethereum

Optimistic Execution Engine (op-geth)

Repository	https://github.com/ethereum-optimism/reference-optimistic-geth
Version	`a68e5aa189e14fde92cec03c1abd98cc7f0db263`
Types	Golang, Solidity
Platforms	Linux, macOS, Windows

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- Documenting invariants across the `OptimismPortal` bedrock smart contract and its relevant subcomponents such as `ResourceMetering`, etc.
- Documenting smart contract and `op-node` invariants across the `op-node` subproject within the `optimism` monorepo.
- Documenting invariants across the `op-geth` project.
- Verification of various invariants throughout unit, integration, and property tests, including:
    - Testing of roundtrip serialization of objects across `op-node` and `op-geth` did not result in the discovery of any new vulnerabilities.
    - Verification of L1/L2 gas fee computation revealed a concern that the `GasPriceOracle` may be misconfigured in a way that locks L2 transaction submission due to unreasonably high transaction fees (TOB-OPTEST-1).
    - Block production and fee computations were not tested for all potential configuration permutations of `op-node` and `op-geth`, but were not found to be problematic throughout tests run during the engagements.
    - Access controls successfully prevented deposit transactions from being submitted over L2 RPC.
    - Data structures such as `go-ethereum` transactions (including the new deposit transaction type) and `BatchData` in the op-node were able to be encoded/decoded successfully without data loss in fuzz tests.
    - Attempting to transfer more ETH in L2 than an account owner holds resulted in errors as expected, while transferring less than an account owner holds resulted in the transfer of requested ETH.
    - OptimismPortal's deposit routines and inherited contract methods behave as expected in terms of burning ETH, hashing, constructing proofs, aliasing addresses for deposits, enforcing gas metering, and more.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Additional testing of system interactions in a concurrent fashion (cascading deposits/withdrawals asynchronously to ensure the state machine behaves as expected).
- Not all invariants could be documented across the system. We recommend further deriving invariants from any off-chain smart contract tests and following up on additional invariants related to the operation of transaction pools, block construction, P2P, payload attribute derivation, and fork conditions.
- We recommend continuing to write fuzz tests for all existing unit tests which do not have an accompanying fuzz test. This will ensure that additional conditions or values which were hard coded within the unit test undergo additional scrutiny.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation
go test	First party unit- and property-testing framework for golang

Test Results

The enumerated properties results of this focused testing are detailed below.

contracts-bedrock

This section details property tests written for the `contracts-bedrock` project located in the `optimism` monorepo under the `packages/contracts-bedrock/` directory.

OptimismPortal: This section details security invariants drawn from the `OptimismPortal` smart contract.

Property	Test	Result
`initialize()` cannot be called more than once	**Property Tests (echidna):** echidna_never_initialize_twice	Passed

The contract cannot be deployed with an invalid `L2OutputOracle` contract address (such as the zero address) and continue to function as intended.	-	Not Tested
The amount of ETH taken by `depositTransaction()` should always equal or exceed the amount signalled to be minted in L2.	**Property Tests (echidna):** echidna_mint_less_than_taken	Passed
A non-zero `_to` address cannot be supplied to `depositTransaction()` when `_isCreation` is set to `true`.	**Property Tests (echidna):** echidna_never_nonzero_to_creation_deposit **Unit Tests (slither):** test_deposit_transaction_integrity	Passed
Gas metering will always burn at least the gas cost calculated from the `_gasLimit` argument when calling `depositTransaction()`.	-	Not Tested
The `from` parameter in the `TransactionDeposited` event emitted by `depositTransaction()` should be aliased if the caller is a contract address.	**Property Tests (echidna):** echidna_alias_from_contract_deposit	Passed
The `from` parameter in the `TransactionDeposited` event emitted by `depositTransaction()` should not be aliased if the caller is an externally-owned address.	**Property Tests (echidna):** echidna_no_alias_from_EOA_deposit	Passed
Calling `L1CrossDomainMessenger.sendMessage`	-	Not Tested

should be equivalent to calling `depositTransaction` with similar parameters directly.		
Calls to `finalizeWithdrawalTransaction` cannot re-enter `finalizeWithdrawalTransaction`.	-	**Not Tested**
A withdrawal cannot be finalized until after the finalization period has concluded.	-	**Not Tested**
A withdrawal can be finalized at most once.	-	**Not Tested**
Withdrawal finalization fails if the L2 Oracle has no output root for the relevant block number.	-	**Not Tested**
Finalization fails if the expected output root cannot be generated from the provided proof.	-	**Not Tested**
Finalization fails if the withdrawal request isn't accompanied by a valid inclusion proof.	-	**Not Tested**
At least `_tx.gasLimit` + `FINALIZE_GAS_BUFFER` gas (weak lower bound) is used when calling finalizeWithdrawalTransaction.	-	**Not Tested**

ResourceMetering: This section details security invariants drawn from the `ResourceMetering` smart contract.

Property	Test	Result

Given a block that uses more than TARGET_RESOURCE_LIMIT gas, the basefee used in the immediately next block is greater.	**Property Tests (echidna):** echidna_high_usage_raise_basefee	Passed
Given a block that used less than TARGET_RESOURCE_LIMIT gas, the basefee used in the immediately next block is lesser (or equal to MINIMUM_BASE_FEE).	**Property Tests (echidna):** echidna_low_usage_lower_basefee	Passed
The basefee is never less than MINIMUM_BASE_FEE.	**Property Tests (echidna):** echidna_never_below_min_basefee	Passed
prevBoughtGas does not exceed MAX_RESOURCE_LIMIT.	**Property Tests (echidna):** echidna_never_above_max_gas_limit	Passed
Given 2 or more empty blocks, the reduction in basefee is greater than the reduction for 1 or fewer empty blocks (down to MINIMUM_BASE_FEE).	-	Not Tested
A block's basefee cannot increase by more than a factor of (1+1/BASE_FEE_MAX_CHANGE_DENOMINATOR) times the immediately preceding block's basefee.	**Property Tests (echidna):** echidna_never_exceed_max_increase	Passed
A block's basefee cannot decrease by more than a factor of (1-1/BASE_FEE_MAX_CHANGE_DENOMINATOR) times the immediately preceding block's basefee.	**Property Tests (echidna):** echidna_never_exceed_max_decrease	Passed

L2OutputOracle: This section details security invariants drawn from the `L2OutputOracle` smart contract.

Property	Test	Result
L2 block numbers are monotonically increasing.		Not Tested
A proposal's block number cannot correspond to a timestamp in the future.		
A proposal with an empty output root is invalid.		Not Tested

AddressAliasHelper: This section details security invariants drawn from the `AddressAliasHelper` smart contract.

Property	Test	Result
L1-to-L2 address aliasing is able to encode any address and decode the original address without failure.	**Property Tests (echidna):** echidna_round_trip_aliasing	Passed

Burn: This section details security invariants drawn from the `Burn` smart contract.

Property	Test	Result
A call to `eth` to burn ETH should result in exactly `_value` ETH being removed from the calling contract.	**Property Tests (echidna):** echidna_burn_eth	Passed
A call to `gas` to burn gas should burn at minimum the amount of gas passed as a	**Property Tests (echidna):**	Passed

parameter.	echidna_burn_gas	

Encoding: This section details security invariants drawn from the `Encoding` smart contract.

Property	Test	Result
Versioned nonce encoding and decoding should succeed for all inputs and be inverse operations of each other.	**Property Tests (echidna):** echidna_round_trip_encoding	Passed

Hashing: This section details security invariants drawn from the `Hashing` smart contract.

Property	Test	Result
A call to `hashCrossDomainMessage` should never succeed when an invalid nonce (i.e., with version > 1) is passed as an argument.	**Property Tests (echidna):** echidna_hash_xdomain_msg_high_version	Passed
A call to `hashCrossDomainMessage` with a version 0 nonce should be equivalent to calling `hashCrossDomainMessageV0` directly.	**Property Tests (echidna):** echidna_hash_xdomain_msg_0	Passed
A call to `hashCrossDomainMessage` with a version 1 nonce should be equivalent to calling `hashCrossDomainMessageV1` directly.	**Property Tests (echidna):** echidna_hash_xdomain_msg_1	Passed

op-node

This section details property tests written for the `op-node` project located in the `optimism` monorepo under the `op-node/` directory. All unit and fuzz tests are written for use with `go test`.

Property	Test	Result
Different `op-node` configurations cannot introduce undefined behavior into the system (inability to finalize deposits or withdrawals).	-	Not Tested
L2 block creation should fail if the new L2 block (with a timestamp of L2's current block head's timestamp + BlockTime) has a timestamp less than the L1 origin block it is derived from.	**Property Tests:** FuzzRejectCreateBlockBadTimestamp	Passed
	Unit Tests: TestRejectCreateBlockBadTimestamp	Passed
Logs other than the `TransactionDeposited` log will not have an inadvertent effect on the system.	**Property Tests:** FuzzDeriveDepositsRoundTrip	Passed
	Unit Tests: TestDeriveUserDeposits	Passed
Deposit logs can be encoded and decoded with their original values intact.	**Property Tests:** FuzzDeriveDepositsRoundTrip	Passed
	Unit Tests: TestDeriveUserDeposits	Passed
An incorrectly parsed `TransactionDeposited` log for a single deposit should not affect other deposits' ability to be processed.	-	Not Tested

An unknown `DEPOSIT_VERSION` specified by an `TransactionDeposited` event will be rejected.	**Property Tests:** FuzzDeriveDepositsBadVersion	Passed
Deposits should not be derived from failed transactions in L1.	**Property Tests:** FuzzDeriveDepositsRoundTrip	Passed
	Unit Tests: TestDeriveUserDeposits	Passed
Deposits are not lost in the event of a previous failure.	-	Not Tested
`L1Info` can be derived into a deposit transaction and back without loss of its original values.	**Property Tests:** FuzzParseL1InfoDepositTxDataValid	Passed
	Unit Tests: TestParseL1InfoDepositTxData	Passed
`L1Info` can be derived from invalid length deposit transaction data will fail.	**Property Tests:** FuzzParseL1InfoDepositTxDataBadLength	Passed
	Unit Tests: TestParseL1InfoDepositTxData	Passed
The correct L1 origin is always selected when constructing a L2 block with	-	Not Tested

`createNewL2Block`.		
`BatchData` can be encoded and decoded with their original values intact.	**Property Tests:** FuzzBatchRoundTrip	Passed
	Unit Tests: TestBatchRoundTrip	Passed
`BatchQueue` ignores batches with a timestamp prior to the safe L2 head timestamp when stepping.	-	Not Tested
`BatchQueue` eagerly updates the `BatchQueueOutput` with `BatchData` submitted with consecutive timestamps, after the safe L2 head	**Unit Tests:** TestBatchQueueEager	Passed
`BatchQueue` progress should be open if the previous progress was open and current progress is closed before stepping	**Unit Tests:** TestBatchQueueFull	Passed
`BatchQueue` progress should be closed if previous progress is closed before stepping	**Unit Tests:** TestBatchQueueFull	Passed
Batches are considered invalid if their timestamp is outside of the minimum/maximum L2 time window.	**Unit Tests:** TestValidBatch	Passed
Batches are considered invalid they were tagged with an epoch number which is not the current one.	**Unit Tests:** TestValidBatch	Passed

Batches are considered invalid if they are not a multiple of block time.	**Unit Tests:** TestValidBatch	**Passed**
Batches are considered invalid if they contain a `DepositTx` type transaction	**Unit Tests:** TestValidBatch	**Passed**
Batches are considered invalid if they do not contain any transactions	**Unit Tests:** TestValidBatch	**Passed**
Batches are considered invalid if their epoch hash does not match the current one	**Unit Tests:** TestValidBatch	**Passed**
Batches are dropped if a reset rolled back a full sequence window or the batch timestamp otherwise precedes the safe L2 head.	-	**Not Tested**

op-geth

This section details property tests written for the `op-geth` project. Some tests exist within the `op-geth` repository directly, while others exist in `op-e2e` within the `optimism` monorepo. The location of the tests is tagged below.

Property	Test	Result
`op-geth` configurations with different values for parameters such as sequence windows and other time-durations cannot introduce undefined behavior into the system, such as an inability to finalize deposits or withdrawals.	-	**Not Tested**
The L1 costs set in the `GasPriceOracle` are enforced in L2 transaction fees appropriately.	-	**Not Tested**

L1 fees are appropriately awarded to the `BaseFeeRecipient`.	-	Not Tested
The nonce of a deposit sender should be incremented in L2, regardless of whether an L1 deposit transaction receipt reported a failure status.	**Unit Tests (op-e2e):** TestMintOnRevertedDeposit	Passed
Deposit transactions which fail to transfer ETH in L2 (e.g. insufficient balance) still retain minted tokens.	**Unit Tests (op-e2e):** TestMintOnRevertedDeposit	Passed
The L1 costs set in the `GasPriceOracle` cannot be incorrectly set to values that prevent the GasPriceOracle from being further updated.	**Unit Tests (op-e2e):** TestGasPriceOracleFeeUpdate s	Failed
The L1 costs set in the `GasPriceOracle` cannot be incorrectly set to values that prevent any transactions from being processed in L2.	**Unit Tests (op-e2e):** TestGasPriceOracleFeesL2Lock	Failed
With the addition of the DepositTx type transaction, transaction serialization is not prone to data loss or misinterpretation.	**Property Tests (op-geth):** FuzzTransactionMarshallingRo undTrip	Passed
The L2 sequencer/verifier should not accept `DepositTx` type transactions over RPC.	**Unit Tests (op-e2e):** TestL2SequencerRPCDepositT x	Passed
Do RPC endpoints enforce size limits appropriately when various deposit transactions are included?	-	Not Tested

L2 will appropriately update state such as account balances in the event that L1 encounters a re-org.	-	Not Tested
The L2 output submitter is updated after a L2 block is committed.	**Unit Tests (op-e2e):** TestL2OutputSubmitter	Passed
The L2 output submitter is resilient towards re-orgs.	-	Not Tested
L2 nodes sync blocks from other nodes before they are confirmed on L1.	**Unit Tests (op-e2e):** TestSystemMockP2P	Passed
The transaction pool appropriately enforces the NoTxPool flag and pushes through forced transactions as expected.	-	Not Tested
The transaction pool can continue to operate and standard Ethereum transactions cannot become expired or stale in the transaction pool due to a large burst of forced transactions.	-	Not Tested
Deposit transactions which failed to transfer value in L2 (e.g. due to insufficient balance) will not negatively affect valid deposit transactions.	**Unit Tests (op-e2e):** TestMixedDepositValidity	Passed
Failed withdrawal transactions should not be able to prevent valid withdrawal transactions (end-to-end).	-	Not Tested
Withdrawals which specify an invalid timestamp, such as one for which an L2 output root doesn't exist or is not FINALIZATION_PERIOD seconds old, should be	**Unit Tests (op-e2e):** TestMixedWithdrawalValidity	Passed

rejected.		
The sender, target, message, value, or gasLimit fields cannot be modified in a withdrawal request without failure.	**Unit Tests (op-e2e):** TestMixedWithdrawalValidity	Passed
A failed deposit in L1 that is then re-org'd to be a successful deposit is handled appropriately by L2.	-	Not Tested
Different verifiers should not derive different fees	-	Not Tested

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

Week 3

ID	Title	Type	Severity
1	Misconfigured GasPriceOracle state variables can lock L2	Data Validation	**Undetermined**

Detailed Findings

1. Misconfigured GasPriceOracle state variables can lock L2	
Severity: **Undetermined**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-OPTEST-1
Target: `optimism/packages/contracts/L2/predeploys/OVM_GasPriceOracle.sol`, `op-geth/core/rollup_l1_cost.go`	

Description

When bootstrapping the L2 network operated by `op-geth`, the `GasPriceOracle` contract is pre-deployed to L2 and its contract state variables are used to dictate L1 costs to be charged on L2. Three state variables are used to compute the cost: `decimals`, `overhead`, and `scalar`, which can be updated through transactions sent to the node.

However, these state variables can seemingly be misconfigured to set gas to a price that does not allow any transactions to be processed. For example, setting overhead to the maximum value of a 256-bit unsigned integer will result in subsequent transactions from being accepted.

In an end-to-end test, contract bindings used in op-e2e tests (such as the `GasPriceOracle` bindings used to update the state variables) would no longer be able to make subsequent transactions/updates, as calls to `SetOverhead` or `SetDecimals` resulted in a deadlock. Sending a transaction directly through the RPC client did not produce a transaction receipt that could be fetched.

Recommendations

Short term, consider implementing checks to ensure `GasPriceOracle` parameters can be updated in the event that fee parameters are previously misconfigured. This could be achieved with an exception to `GasPriceOracle` fees when the contract owner is calling methods within it, or by setting a maximum fee cap.

Long term, ensure operational procedures exist to ensure the system is not deployed or otherwise entered into an unexpected state as a result of operator actions.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Running Tests

The following section provides insight into how a developer can execute the tests generated during the course of the engagement.

Echidna Fuzz Tests (Bedrock Contracts):

Echidna is an Ethereum smart contract fuzzer which allows users to write on-chain property tests to verify the expected states of their application.

Git patches for each project target were provided alongside the report containing the tests generated during the course of the assessment. To prepare the environment for fuzz testing, the `optimism/packages/contracts-bedrock/contracts/test` directory was removed, as it contained unlinked libraries which echidna is incompatible with in a default deployment scheme. Additionally, the hardhat and foundry compilation configurations were updated so they do not strip bytecode hash metadata, as it is also required by echidna to match deployed contracts.

To begin running tests:

- Compile the project by invoking hardhat in the `optimism/packages/contracts-bedrock` directory:

  ```
  npx hardhat clean
  
  npx hardhat compile
  ```

- Invoke echidna against a contract containing property tests using the command:

  ```
  echidna-test --contract <contract_name> --crytic-args
  --hardhat-ignore-compile .
  ```

  This will tell echidna to use the previously created compilation and target the provided contract in a fuzzing campaign.

Go Test Tests (op-node, op-e2e, op-geth)

The `go test` command invokes unit, integration, and fuzz tests written using golang's native `testing` package. Unit and fuzz tests were produced for `op-node` and `op-geth`, as can be observed in the Automated Testing table.

- End-to-end tests within the `optimism/op-e2e` directory and `op-node` unit tests within the `optimism/op-node` directory can be run alongside previously existing tests by running the following command from the respective directories:

  ```
  go test -v ./…
  ```

To run individual unit tests, you may instead use the following command:

```
go test -v -run <TestName>
```

- Fuzz tests written for `op-node` and `op-geth` can be run by running the following command from the directory containing the test file:

  ```
  go test -v -fuzz <TestName>
  ```

  Fuzz tests will run until the process is killed or a keyboard interrupt is detected.

C. Recommendations for improving testability

The following section makes a series of recommendations that would benefit the codebase in increasing testability:

Solidity Smart Contract Testing

- The use of on-chain property fuzzers, such as `echidna`, requires property tests to be written in Solidity. However, on-chain property tests cannot access various aspects of the chain state or results.

- Ensure routines within your contract are testable by confirming inputs, state changes and outputs can be captured by another method in the contract. For instance, events emitted cannot be queried on-chain, they can only be verified off-chain.

    - If a test intends to verify values within an emitted event, consider splitting the method into a helper function that returns the values rather than emitting them in an event. The original method may use this helper function to perform the underlying work and later emit the output data in an event itself, while test methods can target the helper directly to verify output methods.

    - For example, split the `OptimismPortal.depositTransaction` logic into a helper method that returns values rather than simply emitting a log, as these values can be validated by a test using the helper method. Alternatively, wrap the event emitting in a separate virtual function that can be overridden by a test contract which derives from `OptimismPortal` so it can capture these values.

- Ensure your contracts can be easily deployed from another contract where possible. Echidna deploys compiled contracts with no constructor arguments and executes transactions against publicly-accessible methods in an attempt to produce state changes.

    - For contracts which take constructor arguments, developers can create a deriving contract that satisfies the constructor arguments with hardcoded values, or create another contract to deploy it with the appropriate constructor arguments used for testing.

    - This requires care when considering your smart contract code composition.

    - For complex contract developments, explore the use of `etheno` alongside echidna.

- Consider integrating your echidna fuzz tests to your CI/CD pipeline, possibly through the use of echidna GitHub Actions.
    - Leverage the `test-limit` configuration variable to limit the duration of the fuzzing campaign in CI.
    - Ensure fuzz tests are run at a regular interval. Passed test results are not indicative of a lack of vulnerability. The constraints required to violate a property test may not be found in one run, but a latent issue may exist that the fuzzer may catch in another.
- Consider detectors or other custom scripts using the slither API. This can plug into slither's detector API to add rules to slither's existing static analysis rules. An example of how to use the slither API to verify the integrity of a codebase can be observed run by running the following command from the `optimism/packages/contracts-bedrock` directory:

  ```
  python3 ./slither_api_example.py
  ```

  As a proof of concept, this script discovers all echidna property tests and the contracts they live within, states which contracts they immediately inherit from, and performs a check against `OptimismPortal.depositTransaction` to ensure no high-level calls were added/removed, and an if-statement exists for `_isCreation` that contains only a `require` statement that compares `_to` to `address(0)`.

  The test against `OptimismPortal` could be simplified by instead checking the source text for specific segments rather checking every AST node and its underlying IR, but the test was written with this level of granularity to show how one can iterate over every statement or expression in a method and detect specific patterns or use of specific variable across multiple expressions.

  The use of the slither API can enable the CI/CD pipeline to catch issues where a developer mistakenly changes a function in a way they should not have, violating some property of a given method. For instance, a script may differentiate internal and external calls to ensure no external calls are performed in a given method.

L1/L2/op-node Testing

- We recommend creating an API which simplifies end-to-end testing by:
    - Providing methods to initialize accounts with different balances in L1 and L2, and provide a simplified test account structure with the key path, private key, `TransactOpts`, alongside other account properties.

- Ensure timeout-based test failures do not fail due to timeouts being set to low. For example, throughout `op-e2e` tests, various statements wait one second for a block to propagate. Increasing timeout may reduce false-positive test failures for slower systems (possibly within CI).
- Add methods to execute actions such as sending a deposit transaction, withdrawal request, creating arbitrary transfer transactions in L1/L2, causing fork conditions in L1/L2, etc. The testing harness could automatically execute these actions and update expected values such as expected balances/nonces in L1/L2 which are automatically asserted at the end of the test, in addition to any conditions the tester asserts within their test immediately.
    - Simulating fork conditions may require support for rolling back previous actions (and their changes to expected values).
    - Ideally, the system should support invocation of these actions in parallel (from goroutines) to simulate typical network behavior. This includes multiple L1 deposits submitted at once, etc.
    - The system should ensure that blocks being produced in tests simulate conditions for multiple Optimism system-related transactions being included in a single block, as well as individually.
    - Consider writing all relevant end-to-end tests with support for being run against different system configurations, such as specifying differing sequence windows and gas fees.
- Ensure the same test can be rerun with differing system configurations easily, this way existing tests can be repurposed to run against various configurations to ensure there is not undefined behavior in the system from some specific edgecase in different system configurations.

- For unit tests which depend on the result of processing certain data and making sure routines succeed or fail as expected, ensure as many permutations of the input data are tested as possible. Review existing unit tests to identify hardcoded values which may be better suited as randomized or fuzzed variables to increase coverage within tests.
    - For example, some unit tests within `op-node` depend on the `MarshalDepositLogEvent` method to produce a deposit event that is used as input into test deposit derivation functions. Reviewing this method, we can see that deposit versions are hardcoded to valid values. Adding flexibility to helper methods by modifying them to accept additional fields (such as deposit version) will more easily enable testing of additional invariants (such

as ensuring deposit logs with an invalid version do not produce derived deposits).