Optimism SafetyChecker

Consensys Diligence

Date	March 2021

Executive Summary

This report presents the results of our engagement with Optimism PBC to perform differential fuzzing on the `OVM_SafetyChecker` smart contract.

The work was performed by Nicholas Ward and Valentin Wuestholz over the course of two person-weeks between March 8th and March 12th, 2021.

The focus of the engagement was `OVM_SafetyChecker.sol` from the Optimism `contracts/` repo at commit hash `606577457191973b46034602f46ddcc130a5c0ac` (SHA-1 hash `72136297d400fcaeae95f985b53b46c9906f818e`). The contract was compiled using Solidity version `0.7.6` with `optimize-runs=200`.

1 SafetyChecker Summary

The `OVM_SafetyChecker` (Safety Checker) implements a single external method, `isBytecodeSafe()`. This pure function accepts an arbitrary abi-encoded bytestring and performs on-chain static analysis, returning a boolean indicating whether or not the bytestring can be considered "safe" bytecode for the Optimism Virtual Machine (OVM).

Briefly, the OVM endeavors to mimic the semantics of the Ethereum Virtual Machine (EVM). In order to allow fraud proofs to invalidate malicious state updates, OVM execution needs to be replayable in the EVM. This means that, given the same prestate, any state-altering contract execution in the OVM must deterministically produce the same state transition when executed in the EVM.

The initial assumption of a matching prestate is much trickier than it might seem. In order to allow the execution of a transaction in a fraud proof to be applied to the same prestate as the original transaction execution on Layer 2, operations that interact with persistent state or execution context must be replaced with some notion of virtualization.

Any contract that is able to reach one of these un-virtualized opcodes could potentially determine *where* it was being run. This would allow the contract to exhibit different behavior during a fraud proof than during normal execution.

The job of the `OVM_SafetyChecker` is to make sure that any contract deployed on the OVM is unable to reach any of these dangerous opcodes. Because a contract's initialization code can generate arbitrary runtime code, this check needs to be performed twice for every contract deployment – once to ensure the initialization code is safe, and once to ensure the safety of the returned runtime code. To minimize the cost of this operation, the `OVM_SafetyChecker` contract is heavily optimized, making it challenging to reason about directly.

2 Engagement Summary

To test the correctness of the `OVM_SafeyChecker`, we utilized Harvey, our in-house greybox fuzzer for smart contracts (see mariachris.github.io/Pubs/FSE-2020-Harvey.pdf for more details). Because of the difficulty of capturing the expected behavior of the contract logically, we employed a differential fuzzing approach.

For every input bytestring generated by Harvey to pass to the `OVM_SafetyChecker`, the same bytestring was passed to a custom Go implementation of the `isBytecodeSafe()` function. The success (or failure) of the function invocation and the result returned from both implementations were compared for each input. The comparison itself was performed using a custom Harvey tracer, which is defined by special hooks run at specific points during every transaction sequence that Harvey generates.

Throughout the engagement, we developed and refined a Go reference implementation, working with the development team to capture the intended behavior of the `OVM_SafetyChecker` and establish a clear definition of "safe" bytecode. Harvey was able to quickly detect several subtle divergences throughout this refinement process. By diagnosing these divergences between the two implementations, we were able to clarify the specification of the Safety Checker and reason about the implications of accepting certain classes of opcodes.

The Go reference implementation is included in Appendix 1, and the important aspects of its behavior are captured in the informal specification below.

The process of systematically reasoning about the specification of the `OVM_SafetyChecker` lead to several insights into the specification itself as well as adjacent components of the Optimism system. While not explicitly within the scope of this engagement, discussion of these concerns and related recommendations are included in Future Safety Concerns and Gas Considerations.

3 Fuzzing Campaign

After the custom Harvey tracer and the reference implementation were sufficiently refined, we ran a number of longer fuzzing campaigns using Harvey. Our final fuzzing campaigns spanned 16 CPU-days (48h on 8 parallel cores) and generated inputs that explored 152,419 different program paths. The fuzzer achieved 100% code coverage of the Solidity implementation. None of these inputs resulted in detectable divergences between the Solidity implementation and our Go reference implementation.

While this method of testing allowed the behavior of the `OVM_SafetyChecker` to be compared to a carefully developed reference implementation for a large number of inputs, it is important to understand the limitations of a differential fuzzing approach.

The following should be carefully considered in assessing the results of this analysis:

- Not every possible execution path could be explored. There is a potentially infinite number of paths (ignoring gas limits).
- The informal nature of the specification that was established and the potential for unforeseen edge cases to arise.
- The possibility of future changes to the EVM invalidating the reachability analysis performed in either or both of the implementations.
- The possibility of explicitly accepted opcodes allowing evasion of the "sandbox" in a fraud proof.
- The possibility of an error in the Go implementation or the fuzzer itself.

4 Informal Specification

Note that this approach also allowed a number of implicit, compiler-generated checks to be identified and documented. However, these low-level checks are omitted from this specification for clarity.

- All opcodes without explicitly defined semantics in the EVM as of the Istanbul Hardfork are considered **banned**, with the exception of the reserved `INVALID` opcode, `0xfe`.
- Additionally, the following opcodes are considered **banned**:

```
ADDRESS
BALANCE
ORIGIN
GASPRICE
EXTCODESIZE
EXTCODECOPY
EXTCODEHASH
BLOCKHASH
COINBASE
TIMESTAMP
NUMBER
DIFFICULTY
GASLIMIT
CHAINID
SELFBALANCE
SLOAD
SSTORE
CREATE
CALL
CALLCODE
DELEGATECALL
CREATE2
STATICCALL
REVERT
SELFDESTRUCT
```

- The following opcodes are considered **stopping** instructions:

```
STOP
JUMP
RETURN
INVALID
```

- Any opcode that is not contained in the data portion of a `PUSHn` instruction and can be encountered either from the beginning of the bytestring or from a `JUMPDEST` opcode without first encountering a **stopping** instruction is considered **reachable**.
- Any **banned** opcode that is also **reachable** results in rejection of the entire bytestring, with one exception:
  - The following two substrings are allowed. The entire sequence must be **reachable** and matched in its entirety (i.e., any partial match is **banned**):
    - A call to the Identity precompile at address `0x04`:

      `CALLER POP PUSH1 0x00 PUSH1 0x04 GAS CALL`

    - A call to the `OVM_ExecutionManager` contract:

      `CALLER PUSH1 0x00 SWAP1 GAS CALL PC PUSH1 0x0E ADD JUMPI RETURNDATASIZE PUSH1 0x00 DUP1 RETURNDATACOPY RETURNDATASIZE PUSH1 0x00 REVERT JUMPDEST RETURNDATASIZE PUSH1 0x01 EQ ISZERO`
- Any bytestring that is not explicitly rejected is accepted.

5 Future Safety Concerns

TL;DR: The current reachability analysis performed by the Safety Checker can be invalidated by future changes to the EVM.

- To prevent unnecessary restrictions on contract constructor arguments, analyze contract initialization code using a function `OVM_SafetyChecker.isBytecodeSafeOrUnreachable()` that matches the current semantics of `isBytecodeSafe()`.
- To minimize the potential for deployed bytecode to become unsafe in the future, analyze a contract's returned runtime code using a function `OVM_SafetyChecker.isBytecodeSafeRuntime()` which rejects *all* banned or unassigned opcodes and does not perform reachability analysis (except eliminating push data).

Unforeseen changes to the EVM pose a significant risk to the future viability of OVM fraud proofs. There are several non-technical factors that suggest that there may be a rapid acceleration in the pace of EVM-level changes in the near future. Recent research efforts have increasingly embraced the EVM as the primary execution environment for Ethereum going forward, suggesting that efforts to improve it are likely to increase. Breaking changes in particular appear increasingly likely given the Ethereum Community's desire to minimize technical debt leading into the ETH2 merge.

While predicting precisely what these changes might be and how they will be implemented is intractable, minimizing the existing assumptions of the Safety Checker can serve to guard against future impacts to the dispute resolution mechanism. Helpfully, the Safety Checker and much of the Optimism infrastructure, including the OVM itself, can be upgraded in response to future EVM changes. What is much more difficult to change is the *state* of the rollup chain. This suggests that the primary concern is any change to the EVM that allows code sequences currently considered safe by the Safety Checker to interact with persistent state or execution context following a future Ethereum hardfork.

Take the following code sequence as an example:

`... JUMP UNASSIGNED SELFDESTRUCT`

If `UNASSIGNED` is later assigned to be a valid target of a `JUMP` instruction, a key assumption made in the reachability analysis done by the Safety Checker would be invalidated. Specifically, the assumption that only a valid `JUMPDEST` can resume execution after the occurrence of a **stopping** opcode would no longer hold. In the code sequence above, this would potentially make the `SELFDESTRUCT` instruction reachable when executed in the EVM.

A more surprising and perhaps more likely example follows. Note that a multi-byte opcode refers to an instruction that takes an immediate argument from the code that follows rather than from the stack. Currently, the only multi-byte opcodes in the EVM are `PUSHn` instructions, but previous Ethereum Improvement Proposals (EIPs) such as EIP-615 have proposed others.

```
... JUMP UNASSIGNED PUSH1 JUMPDEST SEFLDESTRUCT
```

Suppose `UNASSIGNED` is later assigned to be a multi-byte opcode that takes a single immediate argument. As a result, what was previously a `PUSH1` instruction is now data for the new two-byte opcode. What was previously data for the `PUSH1` instruction is now executable code. This makes the `JUMPDEST` previously contained in push data a valid jump target and would potentially make the `SELFDESTRUCT` instruction reachable when executed in the EVM.

Recommendation

In order to guard against the situations discussed above, an increase in the strictness of the Safety Checker is recommended. Specifically, elimination of the Safety Checker's reachability analysis for runtime bytecode should be considered.

Our working assumption is that the reachability analysis performed in the Safety Checker serves two purposes:

1. Prevent rejection of most safe bytecode that contains unsafe instructions within the metadata appended by the Solidity compiler.
2. Prevent rejection of most safe bytecode that contains unsafe instructions within the constructor arguments appended by the contract deployer.

Note that this prevents unjustified rejection of most but not all occurrences of the two situations mentioned above. A more refined definition of reachability might eliminate false positives by allowing bytecode sequences containing unsafe opcodes following a valid `JUMPDEST` if that `JUMPDEST` could be proven not to be the target of any of the `JUMP` instructions in the bytecode. However, this is not a realistic analysis to perform in the EVM, and presumably the incidence of real-world false positives has been deemed acceptable.

While contract metadata can provide useful information to users of a smart contract, similar benefits can be achieved without including the metadata as executable bytecode. A cursory inspection of the current Solidity-OVM compiler indicates that metadata is currently being excluded from compiled contracts. If this is not the case or accepting metadata is deemed to be a priority, it can be included as push data by interspersing `PUSH` instructions throughout the metadata, though this would complicate the encoding and slightly increase the length of the generated bytecode.

Additional restrictions on constructor arguments do present a serious impediment to usability. However, invalidated assumptions about the reachability of initialization bytecode can be more readily corrected than for runtime bytecode. Because constructor bytecode can not be interacted with after its initial execution, there is only a single window of `dispute_period` length during which a particular piece of initialization code can be included in a fraud proof. As long as an updated version of the Safety Checker that rejects any newly-dangerous code is put in use before the beginning of any dispute period that may overlap with the EVM change in question, execution of dangerous initialization code in the EVM can be avoided. Note that other challenges related to successfully proving fraud before and after a change to the semantics of the EVM may require the length of the dispute period to be increased on either side of a hardfork. This would need to be taken into account when determining the necessary timeline for adjusting the definition of safe initialization code.

For these reasons, we recommend splitting the analysis of bytecode at contract deployment. To prevent unnecessary restrictions on contract constructor arguments, analyze contract initialization code using a function `OVM_SafetyChecker.isBytecodeSafeOrUnreachable()` that matches the current semantics of `isBytecodeSafe()`. To minimize the potential for deployed bytecode to become unsafe in the future, analyze a contract's returned runtime code using a function `OVM_SafetyChecker.isBytecodeSafeRuntime()` which rejects *all* banned or unassigned opcodes and does not perform reachability analysis (except eliminating push data).

5.1 Additional Considerations

Note that there could be additional reasons for compilers to include unreachable data in a contract's bytecode that are not considered here. For example, if Solidity were to implement dynamically-sized `immutable` values, it is likely that the compiler would append a variable amount of data to the runtime bytecode and use `CODECOPY` to access the values during execution. This and other cases could be supported in the future if the EVM were to introduce a `BEGINDATA` opcode or similar. The existence of `BEGINDATA` would also likely eliminate the need for the Safety Checker to perform reachability analysis even on contract initialization code.

It is important to consider that the strictness of the bytecode disallowed in the OVM can only decrease. That is, as soon as a particular restriction is loosened, it must be assumed that a malicious or merely curious actor will deploy bytecode that exercises the full extent of the allowance. The deployed runtime bytecode can not be easily removed from the OVM and can potentially exhibit new and dangerous behavior during a fraud proof as a result of unforeseen changes to the EVM. For this reason, we recommend making the analysis performed by the Safety Checker as strict as possible.

It should be explicitly noted what potential changes to the EVM this recommendation does and does not protect against. Banning all unsafe or unassigned opcodes from runtime bytecode does not protect against changes to opcodes currently considered safe that allow them to exhibit unsafe behavior in the EVM. It does protect against changes to the semantics of currently assigned EVM opcodes that invalidate the reachability assumptions made in the Safety Checker (e.g. if `STOP` was suddenly considered a valid jump target). It also protects against changes to the EVM that assign previously unassigned opcodes either dangerous or "reachability-changing" semantics.

6 Gas Considerations for Fraud Proofs

A critical requirement of the Optimism system is equivalent gas metering between the EVM and the OVM. Any deviation in gas metering between the two can lead to a valid fraud proof being rejected or an invalid fraud proof being accepted, both of which would constitute a consensus failure.

The state transition function of the OVM applies some transaction `T` to the current state `S1`, resulting in a poststate `S2`. To prove that a particular state transition was fraudulent, a verifier submits a fraud proof to the `OVM_FraudVerifier` contract on Layer 1 Ethereum. The transaction in question is then executed in the EVM to determine an expected postate. If this fraud proof determines that transaction `T` applied to prestate `S1` does not in fact result in poststate `S2`, the transaction is considered fraudulent and the head of the Layer 2 blockchain is rolled back to the prestate `S1`.

This strict requirement that the poststate determined in the EVM match the poststate determined in the OVM for any given pair `(S1, T)` imposes a similarly strict requirement on gas metering. Specifically, the gas consumed in both execution environments must also match for all `(S1, T)`. If this is not the case, then there exists a state transition for which execution in the OVM succeeds and execution in the EVM runs out of gas, or vice-versa.

Approaches to reduce the risk of gas-related consensus issues can be divided into two categories:

1. Weaken the requirements on gas metering as much as possible
2. Ensure the requirements on gas metering are met

6.1 Weaken gas metering requirements

Currently, the `GAS` opcode exists un-modified in the OVM. While this allows the use of many common smart contract patterns in OVM contracts, it further strengthens the requirements on gas metering.

Using the `GAS` opcode, a contract can directly measure the gas consumed by any single instruction or sequence of instructions. Given a known deviation in gas metering between the EVM and the OVM for a particular operation, a malicious contract could observe the cost of that operation and use this information to determine its current context.

Because of this, inclusion of the `GAS` opcode strengthens the requirement that gas metering be consistent across any transaction to a requirement that gas metering be consistent across every operation in a transaction.

Recommendation

Consider adding `GAS` to the set of **banned** instructions. Optionally, the `GAS` opcode could be replaced by a function `OVM_ExecutionManager.ovmGAS()` that implements a modified version of the analogous opcode.

Preventing direct measurement of the gas deducted for an operation allows for future modification to the implementation of gas metering in the OVM. For example, divergent costs for specific operations between the EVM and the OVM could be made up for by determining some upper bound on gas consumed by a particular execution and burning any remaining gas at the end. Gas costs could also be deducted at different points during execution as long as the net gas cost for a transaction is unchanged.

6.2 Ensure gas metering requirements are met

A simplified view of *how* the Optimism system ensures the equivalence of gas metering between execution environments consists of a simple basis and a single step:

a) The execution of a given transaction must *begin* with the same amount of gas in both environments.

b) Execution of the same transaction must *consume* the same amount of gas in both environments.

If both of these hold, then the transaction will either run out of gas in both environments, or it will succeed and return the same amount of gas to the transaction origin in both cases.

As discussed above, the existence of the `GAS` opcode requires a more fine-grained property to be satisfied. The exposure of gas metering to contracts means that the amount of gas remaining at each step of execution must be the same in order for the gas behavior to match. To ensure this, the execution must *begin* with the same amount of gas and *consume* the same amount of gas *at each step*.

Ensuring that a transaction begins with the same amount of gas in a fraud proof as in the original OVM execution is complicated by the “all but one 64th” gas semantics of `CALL`. However, this is reasonably well understood and should be possible to guarantee as long as updates are made for future increases in Layer 1 gas costs.

What follows is a discussion of current and future challenges in ensuring that execution consumes the same amount of gas in both the EVM and the OVM.

Current challenges

Currently, the OVM utilizes the `OVM_ExecutionManager` contract to initiate and manage execution. State-changing operations are implemented as calls to the Execution Manager, which then interacts with the State Manager to perform most state modifications. In the EVM, the State Manager exists as a contract, `OVM_StateManager`. In the OVM, calls to the State Manager are intercepted and passed instead to a native Go implementation which interacts directly with the StateDB.

While this pattern avoids much complexity and inefficiency in other areas, it makes meeting the gas metering requirements of the Optimism system particularly challenging. For each interaction with the State Manager, the Go implementation used in the OVM must precisely determine the amount of gas consumed by the same interaction with the `OVM_StateManager` bytecode in the EVM. The cost of these interactions with the `OVM_StateManager` are in some cases dependent on the amount of gas remaining, the progression of a particular portion of the state throughout a transaction, and other complex factors. As discussed in the following section, these pricing functions are likely to become more complex in the near future, and may even become manipulable by external actors in unpredictable ways.

Acknowledging the tradeoffs involved in making such a change, consider implementing a more conservative method for estimating gas costs of the State Manager. It’s possible that replacing the Go implementation completely or running a parallel interaction with the bytecode implementation to measure gas costs could reduce the possibility of gas-related divergences.

Future concerns

As the state size of Layer 1 Ethereum grows, the Ethereum community is likely to continue increasing the costs of state-accessing opcodes while attempting to maintain as much backwards compatibility as possible. This means, among other things, continued increases in the complexity of gas metering for certain instructions. The efforts to avoid breaking existing contracts in particular seems to nudge the trajectory of these changes towards increasingly “stateful” pricing functions. This poses a significant challenge for a protocol that is reliant on deterministic, context-independent execution for fraud proofs.

One immediate concern related to gas-metering changes is the inclusion of EIP-2929 and EIP-2930 in the upcoming Berlin Hardfork.

In short, EIP-2929 introduces transaction-wide access lists and makes the pricing of certain operations dependent on whether or not the addresses and storage slots involved have already been interacted with in the same transaction. Note that what follows is a simplification of the EIP.

Each of the following opcodes is assigned a `cold_cost` and a `warm_cost`*:

```
BALANCE
EXTCODESIZE
EXTCODECOPY
EXTCODEHASH
SLOAD
CALL
CALLCODE
DELEGATECALL
STATICCALL
```

* SELFDESTRUCT and SSTORE are also impacted by this EIP

The `warm_cost` is assessed for any interaction with an address or storage slot that is already in the access lists. The `cold_cost` is assessed for any interaction with an address or storage slot that is not contained in the access lists. After assessing `cold_cost`, the address or storage slot in question is added to the access lists.

In order to avoid breaking existing contracts that rely on the current costs of these operations, EIP-2930 provides a way for the submitter of a transaction to populate the access lists before beginning transaction execution.

EIP-2929 breaks an existing invariant in the way gas costs are charged and allows an attacker to craft transactions that result in non-deterministic fraud proofs. Currently, the cost of all opcodes is dependent only on the local execution context. EIP-2929 makes the cost of some opcodes dependent on a global context that

can not be directly observed at the smart contract level.

With gas metering dependent on this transaction-level access list, it is no longer possible to rely on direct execution of all opcodes in the EVM to deduct the same amount of gas as the matching execution step in the OVM. The submitter of a fraud proof can exert some level of control over the gas costs charged for these opcodes by selectively forcing state accesses into a “warm” mode before beginning the fraud proof.

Note that EIP-2930 is relevant in this case because it negates the possibility of forcing access lists to be empty at the beginning of execution simply by restricting execution of fraud proofs to externally-owned accounts.

Some potential solutions include:

- Force every relevant operation to charge `warm_cost` by measuring the gas deducted and forcing an exception if `cold_cost` is charged.
    - This would mean only charging `warm_cost` in the OVM and forcing all submitters of fraud proofs to provide a complete access list according to `EIP-2930`.
    - This involves significant code complications and assumptions about gas costs, and it may create unforeseen complexity given the need of the Execution Manager and State Manager to use contract storage to persist information about the current execution context.
    - There may be situations where it is not possible to determine which cost was assessed due to overlap with other inputs to the pricing function of an operation that can not be observed at the contract level.
- Propose a new EIP to add an opcode for resetting or interacting with these access lists in the EVM.
    - For example, add an opcode `COLDCALL` that makes a standard call and sets an environment flag similar to `STATICCALL`. `COLDCALL` forces all relevant operations executed in the call and all child calls to ignore the access lists and charge `cold_cost`. This would mean always charging `cold_cost` for these operations in the OVM.
    - Alternatively, add an opcode `WARMCALL`, which executes a call that fails in an exceptional state on any attempt to interact with an address or storage slot not in the access lists.

6.3 Additional Considerations

Precisely matching gas metering in the OVM with the already complex gas semantics of the EVM is a tall task. EIP-2929 serves as an example of how quickly changes to the EVM could make this task more difficult or even impossible without major changes to the system design and implementation.

Wherever possible, drastically simplifying the handling of gas in fraud proofs and in the OVM should be a high priority. As with the acceptance of certain opcodes in the `OVM_SafetyChecker`, once gas-related observations are allowed or more complex gas metering is implemented, it may be difficult to go back on these changes in the future. Active monitoring of the EIP process will also be crucial to understanding the impacts of future EIPs.

Appendix 1 - Go Implementation of the SafetyChecker

```go
package optimism

import (
	"bytes"
	"fmt"
	stdmath "math"
	"math/big"

	"github.com/ethereum/go-ethereum/core/vm"
)

type SafetyChecker interface {
	IsBytecodeSafe(bytecode []byte, value *big.Int) (ret bool, gasMin *big.Int, gasMax *big.Int, err error)
}

type safetyChecker struct {
}

func NewSafetyChecker() SafetyChecker {
	return &safetyChecker{}
}

func (c *safetyChecker) IsBytecodeSafe(bytecode []byte, value *big.Int) (ret bool, gasMin *big.Int, gasMax *big.Int, err error) {
	if value.Sign() != 0 {
		return false, big.NewInt(0), new(big.Int).SetUint64(stdmath.MaxUint64), fmt.Errorf("non-payable")
	}

	ops2Bytes := func(ops []vm.OpCode) []byte {
		var res []byte
		for _, op := range ops {
			res = append(res, byte(op))
		}
		return res
	}

	// Note that these magic call sequences can contain banned opcodes.
	callIdPrecompileSeq := ops2Bytes([]vm.OpCode{
		vm.CALLER, vm.POP, vm.PUSH1, 0x00, vm.PUSH1, 0x04, vm.GAS, vm.CALL})

	callEMSeq := ops2Bytes([]vm.OpCode{
		vm.CALLER, vm.PUSH1, 0x00, vm.SWAP1, vm.GAS, vm.CALL, vm.PC, vm.PUSH1, 0x0E, vm.ADD, vm.JUMPI,
		vm.RETURNDATASIZE, vm.PUSH1, 0x00, vm.DUP1, vm.RETURNDATACOPY, vm.RETURNDATASIZE, vm.PUSH1,
		0x00, vm.REVERT, vm.JUMPDEST, vm.RETURNDATASIZE, vm.PUSH1, 0x01, vm.EQ, vm.ISZERO, vm.PC,
		vm.PUSH1, 0x0a, vm.ADD, vm.JUMPI, vm.PUSH1, 0x01, vm.PUSH1, 0x00, vm.RETURN, vm.JUMPDEST})

	reservedInvalid := vm.OpCode(0xfe)

	// The following instructions may make the next instruction in the bytecode unreachable (until the next JUMPDEST).
	nextPCPotentiallyUnreachableOVM := map[vm.OpCode]bool{
		// This does not include SELFDESTRUCT and REVERT since they are banned.
		vm.STOP:         true,
		vm.JUMP:         true,
		vm.RETURN:       true,
		reservedInvalid: true,
	}

	isValidInEVM := map[vm.OpCode]bool{
		vm.STOP:           true,
		vm.ADD:            true,
```

```
            vm.MUL:           true,
            vm.SUB:           true,
            vm.DIV:           true,
            vm.SDIV:          true,
            vm.MOD:           true,
            vm.SMOD:          true,
            vm.EXP:           true,
            vm.NOT:           true,
            vm.LT:            true,
            vm.GT:            true,
            vm.SLT:           true,
            vm.SGT:           true,
            vm.EQ:            true,
            vm.ISZERO:        true,
            vm.SIGNEXTEND:    true,
            vm.AND:           true,
            vm.OR:            true,
            vm.XOR:           true,
            vm.BYTE:          true,
            vm.SHL:           true,
            vm.SHR:           true,
            vm.SAR:           true,
            vm.ADDMOD:        true,
            vm.MULMOD:        true,
            vm.SHA3:          true,
            vm.ADDRESS:       true,
            vm.BALANCE:       true,
            vm.ORIGIN:        true,
            vm.CALLER:        true,
            vm.CALLVALUE:     true,
            vm.CALLDATALOAD:  true,
            vm.CALLDATASIZE:  true,
            vm.CALLDATACOPY:  true,
            vm.CODESIZE:      true,
            vm.CODECOPY:      true,
            vm.GASPRICE:      true,
            vm.EXTCODESIZE:   true,
            vm.EXTCODECOPY:   true,
            vm.RETURNDATASIZE: true,
            vm.RETURNDATACOPY: true,
            vm.EXTCODEHASH:   true,
            vm.BLOCKHASH:     true,
            vm.COINBASE:      true,
            vm.TIMESTAMP:     true,
            vm.NUMBER:        true,
            vm.DIFFICULTY:    true,
            vm.GASLIMIT:      true,
            vm.CHAINID:       true,
            vm.SELFBALANCE:   true,
            vm.POP:           true,
            vm.MLOAD:         true,
            vm.MSTORE:        true,
            vm.MSTORE8:       true,
            vm.SLOAD:         true,
            vm.SSTORE:        true,
            vm.JUMP:          true,
            vm.JUMPI:         true,
            vm.PC:            true,
            vm.MSIZE:         true,
            vm.GAS:           true,
            vm.JUMPDEST:      true,
            vm.PUSH1:         true,
            vm.PUSH2:         true,
            vm.PUSH3:         true,
            vm.PUSH4:         true,
            vm.PUSH5:         true,
            vm.PUSH6:         true,
            vm.PUSH7:         true,
            vm.PUSH8:         true,
            vm.PUSH9:         true,
            vm.PUSH10:        true,
            vm.PUSH11:        true,
            vm.PUSH12:        true,
            vm.PUSH13:        true,
            vm.PUSH14:        true,
            vm.PUSH15:        true,
            vm.PUSH16:        true,
            vm.PUSH17:        true,
            vm.PUSH18:        true,
            vm.PUSH19:        true,
            vm.PUSH20:        true,
            vm.PUSH21:        true,
            vm.PUSH22:        true,
            vm.PUSH23:        true,
            vm.PUSH24:        true,
            vm.PUSH25:        true,
            vm.PUSH26:        true,
            vm.PUSH27:        true,
            vm.PUSH28:        true,
            vm.PUSH29:        true,
            vm.PUSH30:        true,
            vm.PUSH31:        true,
            vm.PUSH32:        true,
            vm.DUP1:          true,
            vm.DUP2:          true,
            vm.DUP3:          true,
            vm.DUP4:          true,
            vm.DUP5:          true,
            vm.DUP6:          true,
            vm.DUP7:          true,
            vm.DUP8:          true,
            vm.DUP9:          true,
            vm.DUP10:         true,
            vm.DUP11:         true,
            vm.DUP12:         true,
            vm.DUP13:         true,
            vm.DUP14:         true,
            vm.DUP15:         true,
            vm.DUP16:         true,
```

```
		vm.DUP16:        true,
		vm.SWAP1:        true,
		vm.SWAP2:        true,
		vm.SWAP3:        true,
		vm.SWAP4:        true,
		vm.SWAP5:        true,
		vm.SWAP6:        true,
		vm.SWAP7:        true,
		vm.SWAP8:        true,
		vm.SWAP9:        true,
		vm.SWAP10:       true,
		vm.SWAP11:       true,
		vm.SWAP12:       true,
		vm.SWAP13:       true,
		vm.SWAP14:       true,
		vm.SWAP15:       true,
		vm.SWAP16:       true,
		vm.LOG0:         true,
		vm.LOG1:         true,
		vm.LOG2:         true,
		vm.LOG3:         true,
		vm.LOG4:         true,
		vm.CREATE:       true,
		vm.CALL:         true,
		vm.RETURN:       true,
		vm.CALLCODE:     true,
		vm.DELEGATECALL: true,
		vm.CREATE2:      true,
		vm.STATICCALL:   true,
		vm.REVERT:       true,
		vm.SELFDESTRUCT: true,
	}

	isValidInOVM := map[vm.OpCode]bool{}
	for op, valid := range isValidInEVM {
		validOVM := valid
		// We exclude the 25 banned opcodes.
		switch op {
		case
			vm.ADDRESS,
			vm.BALANCE,
			vm.ORIGIN,
			// CALLVALUE is not banned since "it should deterministically be 0 for all OVM call frames".
			// "There are no CALLs with value in the execution manager, and the safety checker enforces that the input
			// to CALL is 0x00. It is banned at the compiler level at the moment, though."
			vm.GASPRICE,
			vm.EXTCODESIZE,
			vm.EXTCODECOPY,
			vm.EXTCODEHASH,
			vm.BLOCKHASH,
			vm.COINBASE,
			vm.TIMESTAMP,
			vm.NUMBER,
			vm.DIFFICULTY,
			vm.GASLIMIT,
			vm.CHAINID,
			vm.SELFBALANCE,
			vm.SLOAD,
			vm.SSTORE,
			vm.CREATE,
			vm.CALL,
			vm.CALLCODE,
			vm.DELEGATECALL,
			vm.CREATE2,
			vm.STATICCALL,
			vm.REVERT,
			vm.SELFDESTRUCT:
			validOVM = false
		}
		if validOVM {
			isValidInOVM[op] = true
		}
	}

	reachable := true
	bytecodeLen := uint64(len(bytecode))
	for pc := uint64(0); pc < bytecodeLen; {
		op := vm.OpCode(bytecode[pc])

		if vm.PUSH1 <= op && op <= vm.PUSH32 {
			// We skip over pushed data.
			// This matches the behavior for validating jump destinations specified in the YP (Sect. 9.4.3).
			// See also https://consensys.net/diligence/blog/2019/12/destroying-the-indestructible.
			numPushed := uint64(op) - uint64(vm.PUSH1) + uint64(1)
			pc += numPushed + uint64(1)
			continue
		}

		if reachable {
			// We check that it is a safe EVM opcode.
			isReservedInvalid := op == reservedInvalid
			isSafeEVMOp := isValidInEVM[op] || isReservedInvalid
			if !isSafeEVMOp {
				return false, big.NewInt(0), new(big.Int).SetUint64(stdmath.MaxUint64), nil
			}

			// We check that it is a safe OVM opcode.
			isSafeOVMOp := isValidInOVM[op] || isReservedInvalid
			if !isSafeOVMOp {
				return false, big.NewInt(0), new(big.Int).SetUint64(stdmath.MaxUint64), nil
			}

			if nextPCPotentiallyUnreachableOVM[op] {
				// If an opcode potentially makes the next instruction unreachable we switch to "unreachable mode"
				// until the next JUMPDEST is encountered.
				reachable = false
			}

			// We can only use CALLER in two magic sequences.
```

```
                if op == vm.CALLER {
                        callIdPrecompileSeqLen := uint64(len(callIdPrecompileSeq))
                        if bytecodeLen < pc+callIdPrecompileSeqLen {
                                return false, big.NewInt(0), new(big.Int).SetUint64(stdmath.MaxUint64), nil
                        }

                        if bytes.Equal(bytecode[pc:pc+callIdPrecompileSeqLen], callIdPrecompileSeq) {
                                // This starts a call to the id precompile. The entire code sequence is valid.
                                pc += callIdPrecompileSeqLen
                                continue
                        }

                        callEMSeqLen := uint64(len(callEMSeq))
                        if bytecodeLen < pc+callEMSeqLen {
                                return false, big.NewInt(0), new(big.Int).SetUint64(stdmath.MaxUint64), nil
                        }

                        if bytes.Equal(bytecode[pc:pc+callEMSeqLen], callEMSeq) {
                                // This starts a call to the ExecutionManager. The entire code sequence is valid.
                                pc += callEMSeqLen
                                continue
                        }

                        // Otherwise, CALLER is a banned operation.
                        return false, big.NewInt(0), new(big.Int).SetUint64(stdmath.MaxUint64), nil
                } else if op == vm.JUMPDEST {
                        reachable = true
                }

                pc++
        }

        // No invalid operations were found.
        return true, big.NewInt(0), new(big.Int).SetUint64(stdmath.MaxUint64), nil
}
```