Arbitrum Stylus

Security Assessment

June 10, 2024

Prepared for:
Harry Kalodner, Rachel Bousfield, Lee Bousfield, Steven Goldfeder, and Ed Felten
Offchain Labs

Prepared by: **Gustavo Grieco, Jaime Iglesias, Kurt Willis, Nat Chin, Dominik Czarnota, Benjamin Samuels, and Troy Sargent**

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.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

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

About Trail of Bits 1
Notices and Remarks 2
Table of Contents 3
Project Summary 5
Executive Summary 7
Project Goals 9
Project Targets 10
Project Coverage 12
Codebase Maturity Evaluation 16
Summary of Findings 20
Detailed Findings 23
1. Gas for WASM program activation not charged early enough 23
2. Project contains no build instructions 25
3. WASM Merkleization is computationally expensive 26
4. WASM binaries lack memory protections against corruption 27
5. Ink is charged preemptively for reading and writing to memory 29
6. Integer overflow vulnerability in brotli-sys 31
7. Reliance on outdated dependencies 32
8. WASM validation relies on Wasmer code that could result in undefined behavior 33
9. Execution of natively compiled WASM code triggers ASan warning 35
10. Unclear program version checks 37
11. Memory leak in capture_hostio 39
12. Use of mem::forget for FFI is error-prone 40
13. Lack of safety documentation for unsafe Rust 42
14. Undefined behavior when passing padded struct via FFI 45
15. Stylus's 63/64th gas forwarding differs from go-ethereum 47
16. Undocumented WASM/WAVM limits 49
17. Missing sanity checks for argumentData instruction 51
18. Discrepancy in EIP-2200 implementation 53
19. Tests missing assertions for some errors and values 55
20. Machine state serialization/deserialization does not account for error guards 57
21. Lack of minimum-value check for program activation 60

22. SetWasmKeepaliveDays sets ExpiryDays instead of KeepaliveDays 62
23. Potential nil dereference error in Node.Start 63
24. Incorrect dataPricer model update in ProgramKeepalive, causing lower cost and demand 64
25. Machine does not properly handle WASM binaries with both Rust and Go support 66
26. Computation of internal stack hash uses wrong prefix string 68
27. WASI preview 1 may be incompatible with future versions 70
28. Possible out-of-bounds write in strncpy function in Stylus C SDK 71
29. Insufficient out-of-bounds check in memcpy utility function for ConstString 73
30. Unused and unset timeouts in Arbitrator's JIT code 74
31. New machine hashing format breaks backward compatibility 76
32. Unclear handling of unexpected machine state transitions 79
33. Potential footguns and attack vectors due to new memory model 82
34. Storage cache can become out of sync for reentrant and delegated calls 83
35. Storage cache can be written to in a static call context 89
36. Revert conditions always override user returned status 92
37. CacheManager bids cannot be increased 94
38. The makeSpace function does not refund excess bid value and can be front-run 96
39. Bids do not account for program size 98
40. Incorrect bid check 100
41. MemoryGrow opcode is underpriced for programs with fixed memory 101

A. Vulnerability Categories 103

B. Code Maturity Categories 104

C. Sequence Diagrams 106

D. Code Quality Findings 112

E. Patch That Extends Tests and Assertions 120

F. Toward an Automated Fuzzing Process 123

G. Recommendations for Improving Integration Tests 127

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Jaime Iglesias, Consultant
jaime.iglesias@trailofbits.com

Kurt Willis, Consultant
kurt.willis@trailofbits.com

Nat Chin, Consultant
natalie.chin@trailofbits.com

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Dominik Czarnota, Consultant
dominik.czarnota@trailofbits.com

Benjamin Samuels, Consultant
benjamin.samuels@trailofbits.com

Project Timeline

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

Date	Event
October 2, 2023	Pre-project kickoff call
October 10, 2023	Status update meeting #1
October 16, 2023	Status update meeting #2
October 20, 2023	Status update meeting #3
November 3, 2023	Status update meeting #4
November 15, 2023	Status update meeting #5
December 4, 2023	Status update meeting #6

December 21, 2023	Status update meeting #7
January 16, 2024	Status update meeting #8
January 22, 2024	Status update meeting #9
March 7, 2024	Pre-project kickoff call for the Stylus interop layer
March 18, 2024	Status update meeting #10
March 25, 2024	Status update meeting #11
April 1, 2024	Status update meeting #12
April 8, 2024	Status update meeting #13
April 15, 2024	Status update meeting #14
April 22, 2024	Delivery of report draft
April 22, 2024	Report readout meeting
June 10, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Offchain Labs engaged Trail of Bits to review the security of Stylus. Stylus is an upgrade to Arbitrum Nitro that enables a new way to write smart contracts by introducing a second, coequal virtual machine (VM) that is fully interoperable with the EVM.

A team of six consultants conducted the review from October 30, 2023, to May 3, 2024, for a total of 47 engineer-weeks of effort. Our testing efforts focused on the Stylus VM and its associated smart contracts. With full access to source code and documentation, we performed static and dynamic testing of the Stylus codebase, using automated and manual processes. A detailed description of the scoping per component is provided in the Project Coverage section.

Observations and Impact

The Stylus audit revealed a number of issues, mostly of low and informational severity, related to WASM program activation/handling/processing (e.g., TOB-STYLUS-1), EVM compatibility (e.g., TOB-STYLUS-15), and cache handling (TOB-STYLUS-34). In general, we found the code to be very robust, with a small number of corner-case but low-impact bugs. However, we must state that the complexity of the code is very high, and it would have been extremely difficult to review without the accompanying documentation, code walkthroughs, and constant communication with the client.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Offchain Labs take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Expand the unit, end-to-end, and random testing.** The codebase needs to be extensively tested with traditional unit tests and smart fuzzing in order to detect unreliable code and potential points of failure. Appendix F and appendix G provide concrete recommendations.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	Count
High	2
Medium	6
Low	10
Informational	21
Undetermined	2

CATEGORY BREAKDOWN

Category	Count
Configuration	1
Data Validation	16
Error Reporting	1
Patching	3
Testing	1
Undefined Behavior	19

Project Goals

The engagement was scoped to provide a security assessment of Arbitrum Stylus. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are WASM programs deterministic after instrumentation?
- Do instrumented WASM programs terminate in a finite number of steps?
- Is there a risk of denial of service?
- Can instrumented WASM programs overflow the host stack?
- Does the middleware accurately track the gas used by user programs?
- Are there differences between the native, JIT, and prover execution modes?
- Are there reachable panics that could cause the node to crash?
- Do the host I/O implementations for the native and prover execution modes diverge in any way?
- Could the use of unsafe Rust and the FFI result in undefined behavior?
- Can users cause excess resource consumption without incurring costs?
- Can error guards be popped outside of user WASM code?
- Are all provided premachine states correctly validated in the one-step prover?
- Is serialization for the one-step prover done correctly?
- Is the price for activating programs applied correctly?
- Is it possible to bypass program expirations to execute programs after they should have expired?
- Does the activation pricing formula produce the intended price?
- Are there any potential denial-of-service vectors due to mispricing?

Project Targets

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

stylus

Repository	https://github.com/OffchainLabs/stylus
Versions	`fc0a69d` (week 1, 2)
	`0509a98` (week 3, 4, 5, 6)
	`3d3cc66` (week 7)
	`233660e` (week 8)
	`1d507c3` (week 9)
	`4d47f3cfb` (week 10, 11)
Individual PRs	PR #208: A host I/O operation for early program exit
	PR #209: Rust-side storage cache
	PR #217: Support for Dencun handling of `SELF_DESTRUCT`
	PR #212: Custom Brotli dictionaries
	PR #220: Compression of modules in the prover
	PR #215: Consolidation of Stylus parameters
	PR #218: Status code simplification
	PR #224: Transient storage host I/O operations
	PR #225: Math host I/O operations
	PR #227: Optimization of function Merkleization
	PR #223, PR #228, PR #41: Cache manager–related PRs
	PR #230: Memory edge cases
	PR #234: Init pricer and mainnet constants
Type	Rust

stylus-contracts

Repository	https://github.com/OffchainLabs/stylus-contracts
Version	`bb98714`

Type	Solidity

wasmer

Repository	https://github.com/OffchainLabs/wasmer
Version	`80125b5`
Type	Rust

stylus-geth

Repository	https://github.com/OffchainLabs/stylus-geth
Version	`27113b8` (week 1, 2)
	`991e082` (week 3, 4)
Type	Go

Project Coverage

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

- **Stylus VM:** Stylus introduces the ability to deploy WASM programs that are executed in a second virtual machine that works alongside the existing EVM. Stylus programs use "ink" (a form of gas), which limits their execution and prevents denial of service. During the engagement, we reviewed the implementation of the different Stylus opcodes, the way ink is charged to Stylus programs, and the behavior of the VM when a program errors out, panics, or runs out of ink.
- **Activation:** WASM programs were not created to run on blockchains, so they lack key features such as gas metering and come with certain operations that should not be allowed on distributed networks. To include gas metering and to ensure that these programs can be safely executed by the network, Stylus programs need to be instrumented (i.e., they need to be parsed to insert gas-metering operations and to validate that they work within the network's rules). This expensive instrumentation and compilation process is called "activation," and it produces WASM programs that can be executed by the network. During our review, we covered the activation process to ensure that gas metering is applied properly, that limitations on user WASM binaries are applied, and that the activation gas cost is charged appropriately; we also checked for different denial-of-service scenarios.
- **Execution modes:** Stylus provides three different execution modes for validators: native, JIT, and prover.
    - Native execution mode uses WASM binaries that are compiled into x86/ARM native code to run them at full speed.
    - In prover mode, users run the same WAVM execution inside the Stylus VM in order to resolve fraud proof challenges. It is the slowest mode, but it offers the highest security guarantees of the three.
    - JIT execution is an intermediate execution mode between native and prover, in which users run the same WASM binaries they would run in native execution mode. It offers higher security guarantees than native execution.
    - Each mode has a different code path inside Stylus and requires user programs to be compiled in different ways. We looked for any divergences in behavior between the modes and looked for corner cases related to the nuances of each mode (e.g., whether a thread could time out in JIT mode). We also verified that user Stylus programs cannot interfere with the main

thread, which performs important operations such as starting user programs and verifying return codes.

- **Fraud proof system:** All the Arbitrum L2 transactions are executed by validators and can be challenged by any user who believes the validators are not honest. During a challenge, the entire block is re-executed in order to determine whether the state is valid and to resolve the challenge, and a proof of the final state is provided. We reviewed the following components of the fraud proof system:
  - Arbitrator: The Arbitrator is a Rust implementation of the Stylus VM that runs when there is a challenge (i.e., it is meant to be run by validators).
  - One Step Proof (OSP): The OSP is a collection of Solidity smart contracts that allow users to emulate a single instruction on Ethereum to prove its execution. It is the lowest level of granularity a challenge can reach.
  - Program linking component: During the execution of fraud proof verification, instead of bisecting the vanilla replay machine's execution of a block, validators bisect execution of user programs, which are dynamically linked in and out over the course of execution, to fraud proof them directly.

  During the review, we looked for any correctness issues in the Arbitrator, such as those that could cause the Arbitrator to force a prover to validate an incorrect state or prevent a prover from validating a correct state. Since part of the fraud proof system is performed on-chain, we looked for Solidity/EVM-related issues and practical limitations that could block the use of the OSP during challenges (e.g., issues that could cause consumption of excessive gas).

- **Host I/O operations:** When a Stylus contract needs to access EVM state, or perform other operations that are not possible in pure WASM, it needs to use the host I/O operations from the Stylus VM for interacting with the contract's environment.

  During the review, we looked for any possible discrepancies between the implementations of host I/O operations in the three execution modes, ensured that host I/O operations are properly priced, and looked for issues that could cause invalid or incorrect data that would impact such operations (e.g., address overflows, dangling pointers, etc.).

- **The memory model:** Stylus introduces a novel VM for which no deployed contracts currently exist. This provided the Stylus developers an opportunity to fix the mistakes of the EVM related to memory; the main difference between Stylus's memory model and the EVM is that memory pricing is global instead of "per call stack."

During the review, we checked whether the memory model is applied properly and looked for edge cases in the new design such as interactions between both models (EVM and Stylus) in the same transaction.

- **Error guards:** At the start of the review, a feature called "error guards" existed. These guards were inserted into user WASM programs during the proving process to allow the proving system to recover from user program errors.

  Later during the review, this feature was removed and reintroduced in a different form with the use of co-threads; co-threads that error out can reenter the main thread at a preestablished program counter. We reviewed the implementation to ensure that it works as intended.

- **Stylus messages, WASI, and co-threads:** WASI allows the Go compiler to target WASM binaries through a standardized API. WASI replaced the previous JavaScript environment in Stylus.

  Because the Go compiler does not support exports, it is not possible for WASM to call back into Go, which is necessary for Stylus programs to execute host I/O operations. (Note that this is only for the JIT and prover execution modes.) To get around this limitation, the Stylus VM uses threads (for JIT execution) and co-threads (for prover execution).

  The basic idea of co-threads is that only a single thread is ever running at a given time (i.e., there is no parallelization), and that thread is executing either the "main thread" or a Stylus program (each call to a Stylus program will create a new thread). Whenever a Stylus program wants to perform a host I/O operation, it has to return a request ID and halt its execution; this request is then picked up by the main thread, which sends the response back to the Stylus program thread and the program resumes execution until it ends.

  The implementation of co-threads differs between the JIT and prover execution modes, but the basic idea is the same. The main difference is that in the JIT execution mode, actual system threads are spawned, and the main thread and the threads executing Stylus programs communicate using synchronous channels; on the other hand, in the prover execution mode, there is only a single thread, but each time a Stylus program is called, the user program is linked to the Arbitrator machine (i.e., becomes part of the machine state) and a new value stack and frame stack are created.

  We checked whether co-threads are correctly implemented and considered scenarios such as the ones below to look for corner cases:

  - What happens if a Stylus program calls itself?

    - What happens when `delegatecall` is used?
    - What happens if a Stylus upgrade is made while a Stylus program is linked (is running in a co-thread)?
    - How do co-threads react to errors and panics?
    - Can a Stylus program or the main thread hang?
    - Is there a way to force a channel to time out in JIT mode?
    - Is the computational cost of using channels properly accounted for?

  Finally, we checked whether the differences between the native, JIT, and prover execution implementations result in discrepancies in their execution.

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:

- Most of the WASM code, except the specific Offchain modifications, was out of scope.
- We assumed that the Go and Rust compiler WASM generation code is correct.
- We did not verify that native execution mode behaves the same for x86 and ARM. We performed only some fuzzing testing, mainly on x86. In order to ensure this property, a large differential fuzzing campaign should be performed.
- The Stylus SDK was out of scope.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	While Stylus is not, generally speaking, an arithmetic-heavy codebase, it still contains critical code that uses mathematical operations, such as pointer arithmetic in the host I/O operations and the memory model. All arithmetic operations we observed use saturating math, but they are not always implemented consistently between the execution modes, which could result in high-severity issues if the code fails to behave in the same way. Additionally, not all expected properties and corner cases are properly documented, and the relevant code is not extensively fuzz tested to ensure it is robust.	**Moderate**
Auditing	Our review of the system's smart contracts was very limited in this audit. Despite our limited review, we found that events are emitted for all relevant on-chain contracts; additionally, there are several instances of logging and debugging options in the off-chain components.	**Satisfactory**
Authentication / Access Controls	Most of the Stylus components define permissionless features allowing users to activate or run any program as long as it is valid. These features do, however, introduce a number of chain parameters. These parameters are correctly handled in the ArbOS state configuration, which is accessible only to the chain owner.	**Satisfactory**
Complexity Management	In its current state, the complexity of the Stylus codebase is high. Some of this complexity is inherent to the software stack used (i.e., because it has to be used to prove the execution of arbitrary EVM and WASM user programs).	**Moderate**

	However, some of the complexity stems from its maturity and the state of some of the features of the dependencies that Stylus relies on (e.g., Wasmer and the Go compiler); the Stylus developers had to create workarounds for some of the current shortcomings of these dependencies until they are addressed in future versions. Currently, the following are the two biggest sources of complexity: 1. The existence of a third execution mode (JIT), which uses system threads and synchronous communication to execute Stylus programs. While we understand the reasoning behind the existence of this mode and we do not recommend removing it immediately, it definitely adds a lot of complexity to the code, as it is basically a third execution path with its own particular nuances. We recommend considering removing this execution mode over the long term as the codebase matures. 2. Because the Go compiler does not support exports for WASM to call back into Go, Stylus programs cannot perform host I/O operations directly; instead, they have to rely on co-threads and requests. This adds another layer of complexity due to the back-and-forth communication.	
Decentralization	The Stylus VM does not alter the decentralization characteristics of Arbitrum networks that integrate it.	**Not Considered**
Documentation	During the review, the Offchain Labs team consistently provided us with documentation, in the form of a Notion file, to help us understand the system and the features under review. As expected, given the duration of this review, the system evolved during the engagement; for example, the JavaScript environment was replaced by WASI, which made certain details of the documentation outdated. To facilitate future audits, we recommend that the Offchain Labs team bring the documentation up to date with the current state of the codebase and formalize it in a single source. Additionally, we were provided with a couple of code	**Moderate**

	walkthroughs during the engagement, which we found extremely useful for understanding the system. We recommend developing a public-facing (or, at least, internal) walkthrough for newcomers to understand the system. Finally, there are certain nuances, assumptions, and component properties that need more documentation, such as the way memory/object ownership between Rust and Go works and the limitations of the pointers passed by Stylus programs when executing host I/O operations. These nuances are well known and understood by the Stylus team but need to be written down to be referenced by both reviewers and developers.	
Low-Level Manipulation	The WASM program activation code relies heavily on inserted snippets of WASM instructions in order to make untrusted code reliable to run in validators. Each of the middleware used needs additional documentation to explain in detail (e.g., line by line) how the assembly is used and how it is expected to run.	**Moderate**
Memory Safety and Error Handling	Both Go and Rust are languages that safely manage memory; however, a lot of the Go and Rust code in the codebase bypasses memory safeness in order to perform cross-language calls. This is a potential source of high-severity issues, which are usually very hard to detect through manual review. While static analyzers are usually capable of reasoning about the memory safety of languages such as Go and Rust, this codebase uses unsafe features, which renders the reasoning ineffective. Finally, the use of random testing is limited; it should be expanded, as it seems to be the best tool available to find this kind of issue.	**Further Investigation Required**
Testing and Verification	There are extensive unit tests, such as the ones found in the `system_tests` folder, for most parts of the system; however, the system needs extensive unit tests for scenarios such as corner cases. Additionally, random testing (a.k.a. fuzzing) is sporadically used; we recommend expanding the use of random testing for a more end-to-end approach. More extensive testing recommendations are provided in appendix F and appendix G.	**Moderate**

Transaction Ordering	There are certain instances in which transaction ordering could lead to unintended behavior, such as when placing bids or making space in the long-term cache contract (TOB-STYLUS-38); however, these are expected in this type of system (i.e., on-chain auction). These instances need to be documented so that users are aware of them.	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Gas for WASM program activation not charged early enough	Data Validation	Medium
2	Project contains no build instructions	Testing	Informational
3	WASM Merkleization is computationally expensive	Data Validation	Low
4	WASM binaries lack memory protections against corruption	Undefined Behavior	Low
5	Ink is charged preemptively for reading and writing to memory	Undefined Behavior	Low
6	Integer overflow vulnerability in brotli-sys	Patching	Low
7	Reliance on outdated dependencies	Patching	Informational
8	WASM validation relies on Wasmer code that could result in undefined behavior	Undefined Behavior	Medium
9	Execution of natively compiled WASM code triggers ASan warning	Undefined Behavior	Informational
10	Unclear program version checks	Undefined Behavior	Informational
11	Memory leak in capture_hostio	Undefined Behavior	Informational
12	Use of mem::forget for FFI is error-prone	Undefined Behavior	Undetermined

13	Lack of safety documentation for unsafe Rust	Undefined Behavior	**Informational**
14	Undefined behavior when passing padded struct via FFI	Undefined Behavior	**Undetermined**
15	Stylus's 63/64th gas forwarding differs from go-ethereum	Undefined Behavior	**Low**
16	Undocumented WASM/WAVM limits	Undefined Behavior	**Informational**
17	Missing sanity checks for argumentData instruction	Undefined Behavior	**Informational**
18	Discrepancy in EIP-2200 implementation	Undefined Behavior	**Informational**
19	Tests missing assertions for some errors and values	Error Reporting	**Low**
20	Machine state serialization/deserialization does not account for error guards	Undefined Behavior	**Low**
21	Lack of minimum-value check for program activation	Data Validation	**Informational**
22	SetWasmKeepaliveDays sets ExpiryDays instead of KeepaliveDays	Undefined Behavior	**Medium**
23	Potential nil dereference error in Node.Start	Data Validation	**Informational**
24	Incorrect dataPricer model update in ProgramKeepalive, causing lower cost and demand	Undefined Behavior	**High**
25	Machine does not properly handle WASM binaries with both Rust and Go support	Data Validation	**Low**
26	Computation of internal stack hash uses wrong prefix string	Data Validation	**Informational**

27	WASI preview 1 may be incompatible with future versions	Patching	**Informational**
28	Possible out-of-bounds write in strncpy function in Stylus C SDK	Data Validation	**Medium**
29	Insufficient out-of-bounds check in memcpy utility function for ConstString	Data Validation	**Medium**
30	Unused and unset timeouts in Arbitrator's JIT code	Configuration	**Informational**
31	New machine hashing format breaks backward compatibility	Data Validation	**Informational**
32	Unclear handling of unexpected machine state transitions	Undefined Behavior	**Informational**
33	Potential footguns and attack vectors due to new memory model	Undefined Behavior	**Informational**
34	Storage cache can become out of sync for reentrant and delegated calls	Data Validation	**High**
35	Storage cache can be written to in a static call context	Data Validation	**Low**
36	Revert conditions always override user returned status	Data Validation	**Low**
37	CacheManager bids cannot be increased	Data Validation	**Informational**
38	The makeSpace function does not refund excess bid value and can be front-run	Undefined Behavior	**Informational**
39	Bids do not account for program size	Data Validation	**Informational**
40	Incorrect bid check	Data Validation	**Informational**

41	MemoryGrow opcode is underpriced for programs with fixed memory	Data Validation	**Medium**

Detailed Findings

1. Gas for WASM program activation not charged early enough	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-STYLUS-1
Target: `arbos/programs/programs.go`	

Description

The gas for activating WASM programs is not charged early enough in the activation code to prevent denial-of-service attacks.

WASM activation is a computationally expensive operation that involves decompressing bytecode (figure 1.1).

```
func (p Programs) ActivateProgram(evm *vm.EVM, program common.Address, debugMode
bool) (uint16, bool, error) {
	statedb := evm.StateDB
	codeHash := statedb.GetCodeHash(program)

	version, err := p.StylusVersion()
	if err != nil {
		return 0, false, err
	}
	latest, err := p.CodehashVersion(codeHash)
	if err != nil {
		return 0, false, err
	}
	// Already compiled and found in the machine versions mapping.
	if latest >= version {
		return 0, false, ProgramUpToDateError()
	}
	wasm, err := getWasm(statedb, program)
	if err != nil {
		return 0, false, err
	}
    {...omitted for brevity...}
```

```
}

func getWasm(statedb vm.StateDB, program common.Address) ([]byte, error) {
	{...omitted for brevity...}
	return arbcompress.Decompress(wasm, MaxWasmSize)
}
```

Figure 1.1: WASM program activation–related code in `arbos/programs/programs.go#L84` and `#L233`

However, if Brotli's decompression fails, the user will not be charged for activating the program, which can be expensive.

Exploit Scenario

Eve creates a specially crafted compressed WASM bytecode with a corrupted bit at the end, with the purpose of slowing down the Arbitrum chain. The corrupted bit causes a failure during decompression, allowing her to avoid paying full price for her program, making her attack cheaper than expected.

Recommendations

Short term, charge gas as early as possible during WASM program activation; gas should be charged even if activation fails for any reason.

Long term, review each computationally expensive operation that can be arbitrarily triggered by users to ensure it is properly priced.

2. Project contains no build instructions	
Severity: **Informational**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-STYLUS-2
Target: `README.md`	

Description

The Stylus repository contains information regarding the project, a roadmap, and information regarding gas pricing, but it lacks other essential information. The repository's README should include at least the following:

- Instructions for building the project
- Instructions for running the built artifacts
- Instructions for running the project's tests

Note that the repository contains a makefile with convenient scripts; however, repositories of this size (e.g., involving a lot of dependencies and Git submodules) are often difficult to build even for experienced developers. Therefore, having building instructions and solutions to common build problems would greatly speed up developer onboarding.

Exploit Scenario

Alice, a developer, tries to build the Stylus repository; however, she faces problems building it due to the missing documentation in the README, and she makes a mistake in the procedure that causes the build to fail.

Recommendations

Short term, add the minimum information listed above to the repository's README. This will help developers to build, run, and test the project.

Long term, as the project evolves, ensure that the README is updated. This will help ensure that it does not communicate incorrect information to users.

3. WASM Merkleization is computationally expensive	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-STYLUS-3
Target: `arbitrator/prover/src/machine.rs`	

Description

A WASM binary with a large global table (e.g., `(table (;0;) 1000000 1000000 externref)`) will require a few seconds of computation to iterate over and hash all table elements (figure 3.1).

```
// Merkleize things if requested
for module in &mut modules {
    for table in module.tables.iter_mut() {
        table.elems_merkle = Merkle::new(
            MerkleType::TableElement,
            table.elems.iter().map(TableElement::hash).collect(),
        );
    }

    let tables_hashes: Result<_, _> =
module.tables.iter().map(Table::hash).collect();
    module.tables_merkle = Merkle::new(MerkleType::Table, tables_hashes?);

    if always_merkleize {
        module.memory.cache_merkle_tree();
    }
}
```

Figure 3.1: A Merkle tree of all table elements being generated (`arbitrator/prover/src/machine.rs#L1395-L1410`)

Exploit Scenario

Eve creates a specially crafted WASM binary containing huge global tables, slowing down the chain.

Recommendations

Short term, reduce the number of table elements that a global table can have to speed up the module parsing process. Consider charging ink for this computation based on the number of elements hashed.

Long term, review each computationally expensive operation that can be arbitrarily triggered by users to ensure it is properly priced.

4. WASM binaries lack memory protections against corruption	
Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-4
Target: `arbitrator`	

Description

Arbitrum compiles user program components to WASM to be run on the network. WASM binaries do not feature modern binary protections that are available by default in native binaries; they are missing most of the common memory safety checks and are vulnerable to related attack primitives (figure 4.1). Arbitrum's compilation to WASM could introduce deviations between native and on-chain execution of a user program.

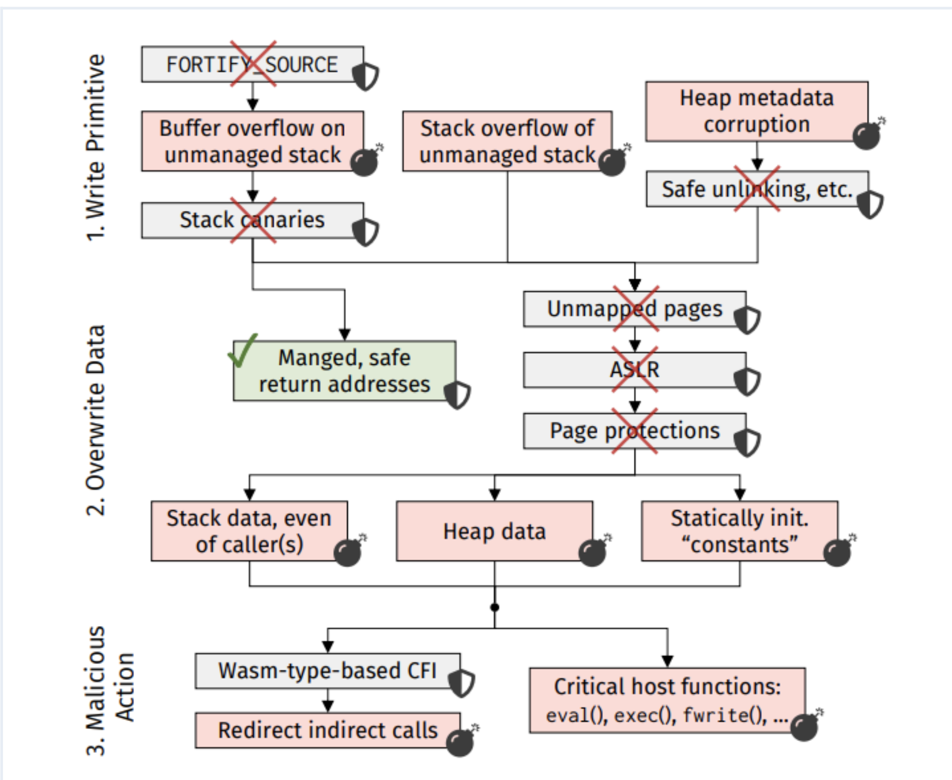


Figure 4.1: An overview of the attack primitives and the missing defenses in the binaries

The USENIX 2020 paper "Everything Old Is New Again: Binary Security of WebAssembly" describes in depth the binary defenses that are missing and new attacks that can be exploited in WASM binaries if memory-unsafe operations are performed.

Other languages provide memory safety in the compiler. Therefore, code that executes safely natively with such checks may not execute the same on-chain.

Exploit Scenario

A user creates a Stylus contract using C/C++ that contains an unsafe memory operation. The user tests the code natively, running it with all the compiler protections enabled, which prevent that operation from being an issue. However, once the user deploys the contract on-chain, an attacker exploits the unsafe memory operation with a shellcode.

Recommendations

Short term, provide documentation advising users to use memory-safe languages. Additionally, advise users to perform extensive testing of any memory-unsafe code that is compiled to WASM to prevent exploitable memory issues.

Long term, review the state of the WASM compiler to evaluate the maturity of its binary protections.

5. Ink is charged preemptively for reading and writing to memory	
Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-5
Target: `arbitrator/stylus/src/host.rs`	

Description

Some host operations for reading and writing to a WASM program's memory charge ink before it is clear whether the operations will be successful or how much ink should really be charged.

For example, in the `read_return_data` function, the user is charged for the operation to write `size` bytes at the start of the host operation. However, the data to be written is the returned data of size `data.len()`, which could actually be smaller than the originally provided `size`. If `data.len()` is smaller than `size`, the user will be charged more ink than they should be.

```
pub(crate) fn read_return_data<E: EvmApi>(
    mut env: WasmEnvMut<E>,
    dest: u32,
    offset: u32,
    size: u32,
) -> Result<u32, Escape> {
    let mut env = WasmEnv::start(&mut env, EVM_API_INK)?;
    env.pay_for_write(size.into())?;

    let data = env.evm_api.get_return_data(offset, size);
    assert!(data.len() <= size as usize);
    env.write_slice(dest, &data)?;

    let len = data.len() as u32;
    trace!("read_return_data", env, [be!(dest), be!(offset)], data, len)
}
```

Figure 5.1: Ink is charged for writing `size` bytes, even though the data to be written could be smaller than `size`. (`arbitrator/stylus/src/host.rs#L273-L289`)

Exploit Scenario

A WASM contract calls `read_return_data`, passing in a very large `size` parameter (100 MB). The EVM API, however, returns only 32 bytes, and the user is overcharged.

Recommendations

Short term, modify the `read_return_data` function to require the user to have enough ink available for writing `size` bytes but to charge ink for writing only `data.len()` bytes. Make similar changes in all host operations that charge ink preemptively.

Long term, review the way ink is charged across different components and levels of abstraction. Make sure it is consistent and follows how the EVM works. Document any discrepancies in the charging of ink.

6. Integer overflow vulnerability in brotli-sys	
Severity: **Low**	Difficulty: **High**
Type: Patching	Finding ID: TOB-STYLUS-6
Target: `arbitrator`	

Description

Running `cargo audit` on the codebase reveals an integer overflow vulnerability in `brotli-sys`, a dependency inherited in the Stylus repository. The dependency does not currently have an update available to fix the vulnerability. Note, however, that the affected functions are not used.

Dependencies should be kept up to date with any fixes to reduce the surface of potentially exploitable code. If no fixes exist for vulnerabilities in dependencies, the relevant area of the code should be clearly documented for developers, including explicit warnings about the vulnerabilities, to ensure that new code does not use vulnerable dependency code.

Exploit Scenario

Alice, an Offchain Labs developer, adds new functionality to the system that uses the `brotli-sys` streaming functions that are affected by the reported vulnerability, introducing an exploitable integer overflow vulnerability into the codebase.

Recommendations

Short term, document the `brotli-sys` streaming functions that are affected by the integer overflow vulnerability with clear warnings for future developers making changes in the system.

Long term, add `cargo audit` to the continuous integration pipeline to ensure that new vulnerabilities are caught quickly. Moreover, continue to monitor dependencies and update them when new versions are available.

7. Reliance on outdated dependencies	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-STYLUS-7
Target: `arbitrator`	

Description

Updated versions of many dependencies of Arbitrum Stylus (and its submodules) are available. Dependency maintainers commonly release updates that contain silent bug fixes, so all dependencies should be periodically reviewed and updated wherever possible.

Dependencies that can be updated are listed in table 7.1, as reported by `cargo upgrade` through the `cargo upgrade --incompatible --dry-run` command.

Dependency	Version Used	Latest Available Version
`thiserror`	`1.0.33`	`1.0.49`
`libc`	`0.2.108`	`0.2.149`
`eyre`	`0.6.5`	`0.6.8`
`sha3`	`0.10.5`	`0.10.18`

Table 7.1: Dependencies in the Stylus repository for which updates are available

Exploit Scenario

Eve learns of a vulnerability in an outdated version of a `sha3` dependency. Knowing that Stylus relies on the outdated version, she exploits the vulnerability.

Recommendations

Short term, update the dependencies to their latest versions wherever possible. Verify that all unit tests pass following such updates. Document any reasons for not updating a dependency.

Long term, add `cargo upgrade --incompatible --dry-run` into the continuous integration pipeline to ensure that new vulnerabilities are caught quickly. Moreover, continue to monitor dependencies and update them when new versions are available.

8. WASM validation relies on Wasmer code that could result in undefined behavior

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-8
Target: `arbitrator/stylus/tools/wasmer/lib/types/src/vmoffsets.rs`	

Description

The use of Wasmer code for validating WASM binaries could result in undefined behavior.

Stylus uses Wasmer to perform a strict validation of WASM binaries before activating them. For instance, the following function computes the memory offset for each WASM binary component:

```
    fn precompute(&mut self) {
        /// Offset base by num_items items of size item_size, panicking on overflow
        fn offset_by(base: u32, num_items: u32, item_size: u32) -> u32 {
            base.checked_add(num_items.checked_mul(item_size).unwrap())
                .unwrap()
        }

        self.vmctx_signature_ids_begin = 0;
        self.vmctx_imported_functions_begin = offset_by(
            self.vmctx_signature_ids_begin,
            self.num_signature_ids,
            u32::from(self.size_of_vmshared_signature_index()),
        );
        self.vmctx_imported_tables_begin = offset_by(
            self.vmctx_imported_functions_begin,
            self.num_imported_functions,
            u32::from(self.size_of_vmfunction_import()),
        );
```

Figure 8.1: The header of the `precompute` function in `lib/types/src/vmoffsets.rs#L282–309`

However, this code relies on unsafe memory operations: it is not guaranteed that the memory pointers are properly aligned, and these pointers can be dereferenced later. Dereferencing of a misaligned memory pointer is undefined behavior (figure 8.2).

```
thread '<unnamed>' panicked at
/home/fuzz/projects/audit-stylus/arbitrator/tools/wasmer/lib/vm/src/instance/mod.rs:
163:18:
```

```
misaligned pointer dereference: address must be a multiple of 0x8 but is
0x51700005066c
```

Figure 8.2: Undefined behavior detected when trying to validate a WASM binary with a misaligned memory pointer

A second, similar issue also exists in the version of Wasmer used by Stylus and can be found by `cargo-careful`, by running `cargo +nightly careful test`.

Exploit Scenario

A user submits a WASM contract that triggers a dereference of a misaligned pointer, which results in a crash or degraded performance.

Recommendations

Short term, modify the associated code to properly align the access pointers to ensure that no undefined behavior is performed. Run related tests in debug mode in the CI pipeline.

Long term, perform fuzz testing of the validation, activation, and execution of WASM contracts. Upgrade to the latest version of Wasmer, which contains fixes for these issues, and integrate `cargo-careful` into the continuous integration pipeline.

9. Execution of natively compiled WASM code triggers ASan warning

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-9
Target: `arbitrator/tools/wasmer/lib/vm/src/libcalls.rs`	

Description

During the execution of natively compiled WASM code, certain code that handles exceptions could produce false positives in the AddressSanitizer (ASan) checks.

Stylus allows users to compile WASM programs into native code and execute it, using Wasmer. While the produced native code looks correct, it seems to be incompatible with certain ASan checks on stack memory:

```
==1584753==WARNING: ASan is ignoring requested __asan_handle_no_return: stack type:
default top: 0x7ffff106a000; bottom 0x7f7d88545000; size: 0x008268b25000
(560102264832)
False positive error reports may follow
For details see https://github.com/google/sanitizers/issues/189
=================================================================
==1584753==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7f7d88546ab0
at pc 0x558196e7d273 bp 0x7f7d88546a90 sp 0x7f7d88546260
WRITE of size 24 at 0x7f7d88546ab0 thread T0
    #0 0x558196e7d272 in sigaltstack
/rustc/llvm/src/llvm-project/compiler-rt/lib/asan/../sanitizer_common/sanitizer_comm
on_interceptors.inc:10100:5
    #1 0x558196eaa3ef in __asan::PlatformUnpoisonStacks()
/rustc/llvm/src/llvm-project/compiler-rt/lib/asan/asan_posix.cpp:45:3
    #2 0x558196eb0417 in __asan_handle_no_return
/rustc/llvm/src/llvm-project/compiler-rt/lib/asan/asan_rtl.cpp:589:8
    #3 0x55819b933581 in
wasmer_vm::trap::traphandlers::raise_lib_trap::h08f8319f19014fcd
/home/fuzz/projects/audit-stylus/arbitrator/tools/wasmer/lib/vm/src/trap/traphandler
s.rs:582:5
    #4 0x55819b94b2c8 in wasmer_vm_memory32_fill
/home/fuzz/projects/audit-stylus/arbitrator/tools/wasmer/lib/vm/src/libcalls.rs:584:
9
    #5 0x7f7f1a400202  (<unknown module>)
    #6 0x7f7f1a40029b  (<unknown module>)

Address 0x7f7d88546ab0 is a wild pointer inside of access range of size
0x000000000018.
SUMMARY: AddressSanitizer: stack-buffer-overflow
/rustc/llvm/src/llvm-project/compiler-rt/lib/asan/../sanitizer_common/sanitizer_comm
on_interceptors.inc:10100:5 in sigaltstack
```

```
Shadow bytes around the buggy address:
  0x7f7d88546800: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7f7d88546880: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7f7d88546900: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7f7d88546980: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7f7d88546a00: 00 00 00 00 00 00 00 00 f1 f1 f1 f1 00 00 00 00
=>0x7f7d88546a80: 00 00 00 f3 f3 f3[f3]f3 00 00 00 00 00 00 00 00
  …
```

Figure 9.1: The header of the ASan warning

While we do not see an immediate risk, the resulting code should be compatible with ASan to make sure the execution of native code can be analyzed.

Recommendations

Investigate the reason for the ASan warning. We were unable to find a recommendation for this issue before the end of the engagement.

10. Unclear program version checks

Severity: **Informational**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-10
Target: `arbos/programs/programs.go`	

Description

When a program is activated, the current Stylus version of the chain is used to compile and instrument the program. If activation is successful, the program state is updated to reflect that version (figure 10.1).

```
programData := Program{
	wasmSize:  wasmSize,
	footprint: info.footprint,
	version:   version,
}
return version, false, p.programs.Set(codeHash, programData.serialize())
```

Figure 10.1: The program version is set in `ActivateProgram`.
(`arbos/programs/programs.go#L210-L215`)

Additionally, as shown in figure 10.2, programs may be reactivated to update the version after Stylus updates; this is useful as instrumentation may change between versions.

```
func (p Programs) ActivateProgram(evm *vm.EVM, program common.Address, debugMode
bool) (uint16, bool, error) {
	statedb := evm.StateDB
	codeHash := statedb.GetCodeHash(program)

	version, err := p.StylusVersion()
	if err != nil {
		return 0, false, err
	}
	latest, err := p.CodehashVersion(codeHash)
	if err != nil {
		return 0, false, err
	}
	// Already compiled and found in the machine versions mapping.
	if latest >= version {
		return 0, false, ProgramUpToDateError()
	}

	// ...
}
```

Figure 10.2: The program version check in `ActivateProgram`
(`arbos/programs/programs.go#L169-L184`)

However, the check in figure 10.2 (`if latest >= version`) implies that a program could have been activated using a Stylus version higher than the current one, which could be the case if the chain's Stylus version is reverted to a previous one after a program is activated; in that case, this check would prevent that program from being reactivated and updated with the current Stylus version.

This behavior in of itself does not necessarily have to be a problem; however, as shown in figure 10.3, a program can be called through the `callProgram` function only when the program's activation version matches the current Stylus version of the chain, which further contradicts the check performed by the activation function.

```
if program.version != stylusVersion {
	return nil, ProgramOutOfDateError(program.version)
}
```

Figure 10.3: The program activation version is checked in `callProgram`.
(`arbos/programs/programs.go#L240-L242`)

Recommendations

Short term, consider whether reactivation of a program should be allowed only when the program's activation version is different from the current Stylus version. This would allow reactivation exclusively when there is a version change; however, note that this might be undesired behavior and should therefore be thoroughly studied.

Long term, document the intended flow for program reactivation under a Stylus version change and consider issues and edge cases that could arise when old programs are reactivated with a different set of instrumentations.

11. Memory leak in capture_hostio	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-11
Target: `stylus/arbitrator/stylus/src/evm_api.rs`	

Description

In the `capture_hostio` function, the `RustBytes` function new calls `mem::forget`, but the allocation is never freed, leaking memory. This may cause excess resource consumption; however, this code appears to be used only when tracing is enabled (presumably in debug mode).

```
fn capture_hostio(&self, name: &str, args: &[u8], outs: &[u8], start_ink: u64,
end_ink: u64) {
    call!(
        self,
        capture_hostio,
        ptr!(RustBytes::new(name.as_bytes().to_vec())),
        ptr!(RustSlice::new(args)),
        ptr!(RustSlice::new(outs)),
        start_ink,
        end_ink
    )
}
```

Figure 11.1: The `capture_hostio` function leaks memory.
(`stylus/arbitrator/stylus/src/evm_api.rs#263–273`)

Recommendations

Short term, have the code explicitly drop `RustBytes`. Alternatively, use `RustSlice`, which `rustc` will automatically free.

Long term, monitor the resource consumption of nodes. For memory managed purely in Rust, run the tests with `cargo miri`.

12. Use of mem::forget for FFI is error-prone

Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-12
Target: `stylus/arbitrator/prover/src/lib.rs`, `stylus/arbitrator/stylus/src/lib.rs`	

Description

The documentation for `std::mem::forget` states that using it to transfer memory ownership across FFI boundaries is error-prone. Specifically, modifications that introduce panics into code that uses `std::mem::forget`, such as the code shown in figures 12.1 and 12.2, may cause double frees, and using a value after calling `as_mut_ptr` and transferring ownership of the memory is invalid. The documentation advises developers to use `ManuallyDrop` instead.

```
pub unsafe extern "C" fn arbitrator_gen_proof(mach: *mut Machine) -> RustByteArray {
    let mut proof = (*mach).serialize_proof();
    let ret = RustByteArray {
        ptr: proof.as_mut_ptr(),
        len: proof.len(),
        capacity: proof.capacity(),
    };
    std::mem::forget(proof);
    ret
}
```

Figure 12.1: The ownership of `proof`'s memory is transferred to `ret`.
(`stylus/arbitrator/prover/src/lib.rs#368–377`)

```
unsafe fn write(&mut self, mut vec: Vec<u8>) {
    self.ptr = vec.as_mut_ptr();
    self.len = vec.len();
    self.cap = vec.capacity();
    mem::forget(vec);
}
```

Figure 12.2: The ownership of `vec`'s memory is transferred to `self`.
(`stylus/arbitrator/stylus/src/lib.rs#84–89`)

Recommendations

Short term, use `ManuallyDrop` instead of `std::mem::forget` in the aforementioned code to more robustly manage memory manually.

Long term, follow best practices outlined in Rust's `stdlib` and test the code thoroughly for leaks and memory corruption.

13. Lack of safety documentation for unsafe Rust

Severity: **Informational**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-13
Target: `arbos/programs/programs.go`	

Description

The Rust codebase's unsafe blocks lack safety comments explaining their invariants and sound usage. Furthermore, safe code and unsafe code are mixed in functions declared unsafe without distinguishing which blocks of code are unsafe. In future versions of Rust, this pattern may be flagged as a warning or even a hard error. Generally, the code would be less ambiguous if unsafe code were explicitly separated into dedicated blocks even if the overall function scope is unsafe.

The following output of running `clippy -- -D clippy::undocumented_unsafe_blocks` shows the unsafe Rust blocks in Stylus that lack documentation on their safety assumptions.

```
...
error: unsafe block missing a safety comment
  --> tools/wasmer/lib/types/src/value.rs:32:29
   |
32 |             .field("bytes", unsafe { &self.bytes })
   |                             ^^^^^^^^^^^^^^^^^^^^^^
   |
   = help: consider adding a safety comment on the preceding line
   = help: for further information visit
https://rust-lang.github.io/rust-clippy/master/index.html#undocumented_unsafe_blocks

error: unsafe block missing a safety comment
  --> tools/wasmer/lib/types/src/value.rs:41:17
   |
41 |                   unsafe { self.$f == *o }
   |                   ^^^^^^^^^^^^^^^^^^^^^^^^
...
```

Figure 13.1: Output of the Clippy linter

Unsafe Rust blocks should always contain safety comments explaining why the unsafe Rust is sound and does not exhibit undefined behavior.

Even if the code is not currently being used in a way that creates undefined behavior, the current Stylus APIs can be used in an unsound manner. Consider the following example:

```
let x = vec![1u8; 1000];
let y = GoSliceData{
    ptr: x.as_ptr(),
    len: 1000
};
let mut a = RustBytes::new(x);
unsafe {
    stylus_vec_set_bytes(&mut a as *mut RustBytes, y);
}
```

Figure 13.2: Example code allowing unsafe behavior

Miri (a tool for detecting undefined behavior) issues a warning on the code in figure 13.2:

```
Undefined Behavior: deallocating while item [SharedReadOnly for <1851>] is strongly
protected by call 812
```

Figure 13.3: Miri's output when run on the code in figure 13.2

The `stylus_vec_set_bytes` function does not contain sufficient documentation that covers this possible unsafe use.

```
///
/// # Safety
///
/// `rust` must not be null.
#[no_mangle]
pub unsafe extern "C" fn stylus_vec_set_bytes(rust: *mut RustBytes, data:
GoSliceData) {
    let rust = &mut *rust;
    let mut vec = Vec::from_raw_parts(rust.ptr, rust.len, rust.cap);
    vec.clear();
    vec.extend(data.slice());
    rust.write(vec);
}
```

Figure 13.4: The `stylus_vec_set_bytes` function (`stylus/arbitrator/stylus/src/lib.rs#201–213`)

The unsafe `write` function also lacks documentation highlighting its possible misuse:

```
unsafe fn write(&mut self, mut vec: Vec<u8>) {
    self.ptr = vec.as_mut_ptr();
    self.len = vec.len();
    self.cap = vec.capacity();
    mem::forget(vec);
}
```

Figure 13.5: The unsafe `write` function (`stylus/arbitrator/stylus/src/lib.rs#84–89`)

The `write` function will leak memory if called consecutively without explicitly freeing vec, such as by using `stylus_vec_set_bytes` (figure 13.6), but this is undocumented.

```
let one = vec![];
let mut two = RustBytes::new(one);
unsafe {
    two.write(vec![1u8; 1000]);
    two.write(vec![1u8; 1000]);
}
```

Figure 13.6: An example of how `write` could be misused

For functions that are called via Cgo, we recommend documenting where memory is allocated and whether the caller is responsible for manually freeing the memory or, in the case of Go, whether it will be garbage collected.

Recommendations

Short term, set the `undocumented_unsafe_blocks`, `unsafe_op_in_unsafe_fn`, and `missing_safety_doc` lints to deny.

Long term, ensure that any implicit assumptions are documented in the code so that they are not forgotten.

14. Undefined behavior when passing padded struct via FFI

Severity: **Undetermined**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-14
Target: `wasmer/lib/vm/src/vmcontext.rs`	

Description

Union types used in Wasmer that cross FFI boundaries and unconditionally transmute between instances of `vmctx` and `host_env` are not derived from `repr(C)`, which could lead to undefined behavior due to inconsistent padding. An example is shown in figures 14.1 and 14.2.

Types that cross FFI boundaries should be derived from `repr(C)` so that "the order, size, and alignment of fields is exactly what you would expect from C or C++," as documented in the Rustonomicon.

```
#[derive(Copy, Clone, Eq)]
pub union VMFunctionContext {
    /// Wasm functions take a pointer to [`VMContext`].
    pub vmctx: *mut VMContext,
    /// Host functions can have custom environments.
    pub host_env: *mut std::ffi::c_void,
}

impl VMFunctionContext {
    /// Check whether the pointer stored is null or not.
    pub fn is_null(&self) -> bool {
        unsafe { self.host_env.is_null() }
    }
}
```

Figure 14.1: A union that is used across FFI boundaries (`wasmer/lib/vm/src/vmcontext.rs#25–38`)

```
/// Call the wasm function pointed to by `callee`.
///
/// * `vmctx` - the callee vmctx argument
/// * `caller_vmctx` - the caller vmctx argument
/// * `trampoline` - the jit-generated trampoline whose ABI takes 4 values, the
///   callee vmctx, the caller vmctx, the `callee` argument below, and then the
///   `values_vec` argument.
/// * `callee` - the third argument to the `trampoline` function
/// * `values_vec` - points to a buffer which holds the incoming arguments, and to
///   which the outgoing return values will be written.
```

```
///
/// # Safety
///
/// Wildly unsafe because it calls raw function pointers and reads/writes raw
/// function pointers.
pub unsafe fn wasmer_call_trampoline(
    trap_handler: Option<*const TrapHandlerFn<'static>>,
    config: &VMConfig,
    vmctx: VMFunctionContext,
    trampoline: VMTrampoline,
    callee: *const VMFunctionBody,
    values_vec: *mut u8,
) -> Result<(), Trap> {
    catch_traps(trap_handler, config, || {
        mem::transmute::<_, extern "C" fn(VMFunctionContext, *const VMFunctionBody,
*mut u8)>(
            trampoline,
        )(vmctx, callee, values_vec);
    })
}
```

Figure 14.2: A call to a foreign interface with the union shown in figure 14.1 (`wasmer/lib/vm/src/trap/traphandlers.rs#642–670`)

Recommendations

Short term, derive types that cross FFI boundaries from `repr(C)`.

Long term, enable Clippy's `default_union_representation` lint and integrate `cargo miri` into the testing of Wasmer.

15. Stylus's 63/64th gas forwarding differs from go-ethereum	
Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-15
Target: `audit-stylus/arbos/programs/api.go`, `audit-stylus/arbitrator/stylus/src/host.rs`	

Description

The Stylus VM deviates from the Ethereum specification and the behavior of the reference implementation in its application of the 63/64th gas forwarding rule, defined in EIP-150.

EIP-150 states that "if a call asks for more gas than all but one 64th of the maximum allowed amount, call with all but one 64th of the maximum allowed amount of gas."

The Go implementation of the Ethereum protocol calculates the "all but one 64th" amount in the `callGas` function. The new rule is applied only when the requested amount of gas exceeds the allowed gas computed using the rule.

```
evm.callGasTemp, err = callGas(evm.chainRules.IsEIP150, contract.Gas, gas,
stack.Back(0))
```

Figure 15.1: go-ethereum's calculation for gas available in CALL (`go-ethereum/core/vm/gas_table.go#391`)

```
func callGas(isEip150 bool, availableGas, base uint64, callCost *uint256.Int)
(uint64, error) {
	if isEip150 {
		availableGas = availableGas - base
		gas := availableGas - availableGas/64
		// If the bit length exceeds 64 bit we know that the newly calculated
"gas" for EIP150
		// is smaller than the requested amount. Therefore we return the new
gas instead
		// of returning an error.
		if !callCost.IsUint64() || gas < callCost.Uint64() {
			return gas, nil
		}
	}
	if !callCost.IsUint64() {
		return 0, ErrGasUintOverflow
	}

	return callCost.Uint64(), nil
```

```
}
```

Figure 15.2: An application of 63/64th rule (`go-ethereum/core/vm/gas.go#37–53`)

On the other hand, Stylus applies the 63/64th rule indiscriminately using the minimum value of the requested gas amount and the gas available to the parent call. The 63/64th rule should be applied only if the call requests more than “all but one 64th” of the gas.

```
gas = gas.min(env.gas_left()?); // provide no more than what the user has

let contract = env.read_bytes20(contract)?;
let input = env.read_slice(calldata, calldata_len)?;
let value = value.map(|x| env.read_bytes32(x)).transpose()?;
let api = &mut env.evm_api;

let (outs_len, gas_cost, status) = call(api, contract, &input, gas, value);
```

Figure 15.3: Stylus’s calculation for gas available in CALL (`arbitrator/stylus/src/host.rs#153–160`)

```
startGas := gas

// computes makeCallVariantGasCallEIP2929 and gasCall/gasDelegateCall/gasStaticCall
baseCost, err := vm.WasmCallCost(db, contract, value, startGas)
if err != nil {
	return 0, gas, err
}
gas -= baseCost

// apply the 63/64ths rule
one64th := gas / 64
gas -= one64th
```

Figure 15.4: Stylus’s incorrect application of the 63/64th rule (`arbos/programs/api.go#114–125`)

Recommendations

Short term, have the code pass all but one 64th of the available gas only if a call requests more than the maximum allowed gas.

Long term, develop machine-readable tests for the Stylus VM that include the expected gas consumption, similar to Ethereum’s reference tests.

16. Undocumented WASM/WAVM limits

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-16
Target: `arbitrator/prover/src/programs/mod.rs`	

Description

When a user WASM program is parsed, certain limits are enforced in the program (figure 19.1). These limits are undocumented, so they might be unexpected for users.

Additionally, while those limits serve as protection against denial of service and extra checks for bugs, not all of the WASM binary fields are explicitly limited. For example, the number of imports (the `imports` field) is not checked, yet this number is constrained by the imports available to the implementation (VM, host modules, etc.).

```
pub fn parse_user(wasm: &'a [u8], page_limit: u16, compile: &CompileConfig) ->
Result<(WasmBinary<'a>, StylusData, u16)> {
    // ...
    // ensure the wasm fits within the remaining amount of memory
    if pages > page_limit.into() {
        let limit = page_limit.red();
        bail!("memory exceeds limit: {} > {limit}", pages.red());
    }

    // not strictly necessary, but anti-DoS limits and extra checks in case of bugs
    macro_rules! limit { ... }
    limit!(1, bin.memories.len(), "memories");
    limit!(100, bin.datas.len(), "datas");
    limit!(100, bin.elements.len(), "elements");
    limit!(1_000, bin.exports.len(), "exports");
    limit!(1_000, bin.tables.len(), "tables");
    limit!(10_000, bin.codes.len(), "functions");
    limit!(50_000, bin.globals.len(), "globals");
    for function in &bin.codes {
        limit!(4096, function.locals.len(), "locals")
    }

    let table_entries = bin.tables.iter().map(|x| x.initial).saturating_sum();
    limit!(10_000, table_entries, "table entries");

    let max_len = 500;
    macro_rules! too_long { ... }
    if let Some((name, _)) = bin.exports.iter().find(|(name, _)| name.len() >
max_len) {
```

```
        too_long!("name", name.len())
    }
    if bin.names.module.len() > max_len {
        too_long!("module name", bin.names.module.len())
    }
    if bin.start.is_some() {
        bail!("wasm start functions not allowed");
    }
```

Figure 16.1: Limits enforced in `parse_user`
(`arbitrator/prover/src/binary.rs#L585-L648`)

Recommendation

Short term, document the limits enforced on parsed WASM programs along with an explanation of how those limits were chosen. Also, consider whether limits for currently unchecked fields such as `imports` should be set.

Long term, benchmark the chosen limits to make sure they do not allow for any denial-of-service scenario.

17. Missing sanity checks for argumentData instruction

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-17
Target: `stylus-contracts/src/osp/OneStepProver0.sol`	

Description

The `argumentData` instruction is missing sanity checks in certain cases.

In most cases, `argumentData` is checked to ensure it does not contain any unexpected and unwanted bits, as is done in the `executeCrossModuleCall` function.

```
// Jump to the target
uint32 func = uint32(inst.argumentData);
uint32 module = uint32(inst.argumentData >> 32);
require(inst.argumentData >> 64 == 0, "BAD_CROSS_MODULE_CALL_DATA");
```

Figure 17.1: A check for unexpected higher bits
(`stylus-contracts/src/osp/OneStepProver0.sol#158–161`)

However, in some cases, such as in the `executeCrossModuleInternalCall` function (figure 17.2), `argumentData` is simply truncated or unchecked.

```
uint32 internalIndex = uint32(inst.argumentData);
uint32 moduleIndex = mach.valueStack.pop().assumeI32();
Module memory calledMod;
```

Figure 17.2: The `argumentData` instruction is truncated.
(`stylus-contracts/src/osp/OneStepProver0.sol#174–176`)

Additionally, the `executeConstPush` function does not check `argumentData` for set upper bits when its value is an `I32`.

```
function executeConstPush(
    Machine memory mach,
    Module memory,
    Instruction calldata inst,
    bytes calldata
) internal pure {
    uint16 opcode = inst.opcode;
    ValueType ty;
    if (opcode == Instructions.I32_CONST) {
        ty = ValueType.I32;
```

```
    } else if (opcode == Instructions.I64_CONST) {
        ty = ValueType.I64;
    } else if (opcode == Instructions.F32_CONST) {
        ty = ValueType.F32;
    } else if (opcode == Instructions.F64_CONST) {
        ty = ValueType.F64;
    } else {
        revert("CONST_PUSH_INVALID_OPCODE");
    }

    mach.valueStack.push(Value({valueType: ty, contents:
uint64(inst.argumentData)}));
}
```

Figure 17.3: The `executeConstPush` function pushes 64 bits of `argumentData` to the value stack. (`stylus-contracts/src/osp/OneStepProver0.sol#38–59`)

However, this case would mean that incorrectly parsed WASM code is being executed, which is unlikely.

Recommendation

Short term, include the missing sanity checks for `argumentData` to ensure the upper bits are not set for small value types. Add these checks in all instances mentioned in the finding as well as any others that are identified.

Long term, consider adding more sanity checks in areas of the code where the security of many parts relies on one assumption.

18. Discrepancy in EIP-2200 implementation

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-18
Target: `stylus/go-ethereum/core/vm/operations_acl_arbitrum.go`	

Description

The `WasmStateStoreCost` function, which is an adaptation of `go-ethereum`'s `makeGasSStoreFunc` function, introduces a discrepancy from the original code (and a deviation from the EIP-2200 specification) when performing EIP-2200's "stipend check."

In particular, as shown in figure 18.2, the "stipend check" is performed in the original code as a "less than or equal to" comparison.

```
func makeGasSStoreFunc(clearingRefund uint64) gasFunc {
	return func(evm *EVM, contract *Contract, stack *Stack, mem *Memory,
memorySize uint64) (uint64, error) {
		// If we fail the minimum gas availability invariant, fail (0)
		if contract.Gas <= params.SstoreSentryGasEIP2200 {
			return 0, errors.New("not enough gas for reentrancy sentry")
		}
```

Figure 18.2: The original `go-ethereum` code
(`go-ethereum/core/vm/operations_acl.go#L27-L30`)

In Stylus's version of the code, the stipend check is meant to be performed by the caller of the function.

```
// Computes the cost of doing a state store in wasm
// Note: the code here is adapted from makeGasSStoreFunc with the most recent
parameters as of The Merge
// Note: the sentry check must be done by the caller
func WasmStateStoreCost(db StateDB, program common.Address, key, value common.Hash)
uint64 {
```

Figure 18.2: The adapted `go-ethereum` code in Stylus
(`stylus/go-ethereum/core/vm/operations_acl_arbitrum.go#L40–L43`)

For example, the check is handled in the `user_host__storage_store_bytes32` function as part of the host operations (figure 18.3).

```
pub unsafe extern "C" fn user_host__storage_store_bytes32(key: usize, value: usize)
{
```

```
    let program = Program::start(2 * PTR_INK + EVM_API_INK);
    program.require_gas(evm::SSTORE_SENTRY_GAS).unwrap();
    [...]
}
```

Figure 18.3: The EIP-2200 "stipend check" performed by the caller (`arbitrator/wasm-libraries/user-host/src/host.rs#L38-L40`)

However, the check is performed as a strictly "less than" comparison, thereby introducing a discrepancy from EIP-2200 and from the code being adapted (note that `require_gas` calls `require_ink`).

```
fn require_ink(&mut self, ink: u64) -> Result<(), OutOfInkError> {
    let ink_left = self.ink_ready()?;
    if ink_left < ink {
        return self.out_of_ink();
    }
    Ok(())
}
```

Figure 18.4: The "stipend check" implementation (`arbitrator/prover/src/programs/meter.rs`)

Note that in this particular case, the deviation does not lead to any security issues.

This discrepancy also appears in the `storage_store_bytes32` function.

Recommendation

Short term, modify the two affected functions so that they perform the stipend check using a "less than or equal to" comparison, per the EIP-2200 specification.

Long term, whenever code is being adapted from a different source, thoroughly document any expected deviations; additionally, adapt the original tests, which can help identify any expected deviations.

19. Tests missing assertions for some errors and values

Severity: **Low**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-STYLUS-19
Target: `stylus/arbitrator/prover/src/binary.rs`	

Description

Many of the tests in the codebase perform incomplete assertions, which may prevent the tests from detecting bugs in the event of future code changes. In particular, some tests check only the following:

- Whether an error was returned, but not the type or the message of the error
- Whether the resulting structure's field values are as expected

Additionally, the tests do not test all edge cases. For example, there are no unit tests that ensure that the enforced WASM limits (mentioned in TOB-STYLUS-16) actually work.

Those issues can be seen, for example, in the prover's tests, as shown in figure 19.1.

```
#[test]
pub fn reject_reexports() {
    let wasm = as_wasm(...);
    let _ = binary::parse(&wasm, Path::new("")).unwrap_err();

    let wasm = as_wasm(...);
    let _ = binary::parse(&wasm, Path::new("")).unwrap_err();
}

#[test]
pub fn reject_ambiguous_imports() {
    let wasm = as_wasm(...);
    let _ = binary::parse(&wasm, Path::new("")).unwrap();

    let wasm = as_wasm(...);
    let _ = binary::parse(&wasm, Path::new("")).unwrap_err();
}
```

Figure 19.1: `stylus/arbitrator/prover/src/test.rs#L14-L54`

Recommendations

Short term, apply the patch provided in appendix E to improve the quality of the tests.

Long term, further refactor the tests to ensure they include assertions for all expected states of values or errors that are returned from the tested functions.

20. Machine state serialization/deserialization does not account for error guards

Severity: **Low**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-20
Target: `stylus/arbitrator/prover/src/machine.rs`	

Description

The code for serialization and deserialization of the machine state does not account for any error guards (figure 20.1). If any error guards are present, they could produce an invalid machine state when the prover is run from a deserialized state.

```
pub fn serialize_state<P: AsRef<Path>>(&self, path: P) -> Result<()> {
    let mut f = File::create(path)?;
    let mut writer = BufWriter::new(&mut f);
    let modules = self
        .modules
        .iter()
        .map(|m| ModuleState {
            globals: Cow::Borrowed(&m.globals),
            memory: Cow::Borrowed(&m.memory),
        })
        .collect();
    let state = MachineState {
        steps: self.steps,
        status: self.status,
        value_stack: Cow::Borrowed(&self.value_stack),
        internal_stack: Cow::Borrowed(&self.internal_stack),
        frame_stack: Cow::Borrowed(&self.frame_stack),
        modules,
        global_state: self.global_state.clone(),
        pc: self.pc,
        stdio_output: Cow::Borrowed(&self.stdio_output),
        initial_hash: self.initial_hash,
    };
    bincode::serialize_into(&mut writer, &state)?;
    writer.flush()?;
    drop(writer);
    f.sync_data()?;
    Ok(())
}

// Requires that this is the same base machine. If this returns an error, it has not
mutated `self`.
```

```rust
pub fn deserialize_and_replace_state<P: AsRef<Path>>(&mut self, path: P) ->
Result<()> {
    let reader = BufReader::new(File::open(path)?);
    let new_state: MachineState = bincode::deserialize_from(reader)?;
    if self.initial_hash != new_state.initial_hash {
        bail!(
            "attempted to load deserialize machine with initial hash {} into machine
with initial hash {}",
            new_state.initial_hash, self.initial_hash,
        );
    }
    assert_eq!(self.modules.len(), new_state.modules.len());

    // Start mutating the machine. We must not return an error past this point.
    for (module, new_module_state) in
self.modules.iter_mut().zip(new_state.modules.into_iter())
    {
        module.globals = new_module_state.globals.into_owned();
        module.memory = new_module_state.memory.into_owned();
    }
    self.steps = new_state.steps;
    self.status = new_state.status;
    self.value_stack = new_state.value_stack.into_owned();
    self.internal_stack = new_state.internal_stack.into_owned();
    self.frame_stack = new_state.frame_stack.into_owned();
    self.global_state = new_state.global_state;
    self.pc = new_state.pc;
    self.stdio_output = new_state.stdio_output.into_owned();
    Ok(())
}
```

Figure 20.1: Machine state serialization and deserialization code (`stylus/arbitrator/prover/src/machine.rs#L1430-L1488`)

When a machine state is serialized and later deserialized—as is the case when `CreateValidationNode` is run (figure 20.2)—the information about any error guards is lost.

```go
func CreateValidationNode(configFetcher ValidationConfigFetcher, stack *node.Node,
fatalErrChan chan error) (*ValidationNode, error) {
```

Figure 20.2: The `CreateValidationNode` function (`stylus/validator/valnode/valnode.go#L87`)

This would result in a mismatch between the actual machine state and that which starts from a serialized state.

Exploit Scenario

Alice creates a validation node from a serialized machine state. Because the error guards were not included during serialization, the correct execution of the machine is now undetermined.

Recommendation

Short term, include `ErrorGuardStack` (`machine.guards`) as part of the machine state serialization and deserialization process.

Long term, when introducing new features, keep in mind all of the areas that might be affected by them and ensure there is sufficient test coverage.

21. Lack of minimum-value check for program activation

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-STYLUS-21
Target: `stylus/precompiles/ArbWasm.go`	

Description

The cost for activating WASM programs is paid in native currency instead of gas. However, there is no check of the supplied native currency at the start of the program activation code. This is presumably because the cost is not known up front; nonetheless, a simple zero-value or minimum-value check could prevent the need to perform unnecessary computation if the user supplies insufficient value.

```
// Compile a wasm program with the latest instrumentation
func (con ArbWasm) ActivateProgram(c ctx, evm mech, value huge, program addr)
(uint16, error) {
	debug := evm.ChainConfig().DebugMode()

	// charge a fixed cost up front to begin activation
	if err := c.Burn(1659168); err != nil {
		return 0, err
	}
	version, codeHash, moduleHash, dataFee, takeAllGas, err :=
c.State.Programs().ActivateProgram(evm, program, debug)
	if takeAllGas {
		_ = c.BurnOut()
	}
	if err != nil {
		return version, err
	}
	if err := con.payActivationDataFee(c, evm, value, dataFee); err != nil {
		return version, err
	}
	return version, con.ProgramActivated(c, evm, codeHash, moduleHash, program,
version)
}
```

Figure 21.1: WASM program activation code (`stylus/precompiles/ArbWasm.go#L24-L43`)

Recommendation

Short term, include a zero-value or minimum-value check at the start of the program activation code.

Long term, review the codebase to identify any other possibly unnecessary computations that could be avoided by checks made in advance.

22. SetWasmKeepaliveDays sets ExpiryDays instead of KeepaliveDays

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-22
Target: `stylus/precompiles/ArbOwner.go`	

Description

The `SetWasmKeepaliveDays` function sets the `ExpiryDays` value instead of the `KeepaliveDays` value, making admins unable to set the `KeepaliveDays` value from the Go side.

```
// Sets the number of days after which programs deactivate
func (con ArbOwner) SetWasmExpiryDays(c ctx, _ mech, days uint16) error {
	return c.State.Programs().SetExpiryDays(days)
}

// Sets the age a program must be to perform a keepalive
func (con ArbOwner) SetWasmKeepaliveDays(c ctx, _ mech, days uint16) error {
	return c.State.Programs().SetExpiryDays(days)
}
```

Figure 22.1: `stylus/precompiles/ArbOwner.go#L200–L208`

Exploit Scenario

An admin makes a call to `SetWasmKeepaliveDays` with the intention of extending the life of some programs; however, they inadvertently expire all programs, as the function incorrectly sets `ExpiryDays`.

Recommendations

Short term, fix the `SetWasmKeepaliveDays` function to properly set `KeepaliveDays` instead of `ExpiryDays` for programs.

Long term, add tests to ensure that the setter and getter functions of chain properties work correctly.

23. Potential nil dereference error in Node.Start	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-STYLUS-23
Target: `stylus/arbnode/node.go`	

Description

The `Node.Start` function may crash the node due to a `nil` dereference error.

A nil dereference error can happen when `Node.Start` calls `n.configFetcher.Get` (figure 23.1). We assume that `n.configFetcher` can be `nil`, as suggested by the `nil` check at the end of the `Node.Start` function. If `n.configFetcher` is `nil`, a `nil` dereference error will occur when `Node.Start` calls the `LiveConfig` type's `Get` method on it (figure 23.2).

We have not determined whether `n.configFetcher` can actually be `nil`.

```
func (n *Node) Start(ctx context.Context) error {
	// config is the static config at start, not a dynamic config
	config := n.configFetcher.Get()

	(...)

	if n.configFetcher != nil {
		n.configFetcher.Start(ctx)
	}
	return nil
}
```

Figure 23.1: `stylus/arbnode/node.go#L999–L1126`

```
func (c *LiveConfig[T]) Get() T {
	c.mutex.RLock()
	defer c.mutex.RUnlock()
	return c.config
}
```

Figure 23.2: `stylus/cmd/genericconf/liveconfig.go#L38–L42`

Recommendation

Short term, verify whether `n.configFetcher` can be `nil` in the `Node.Start` function; if it cannot be `nil`, remove the `nil` check from the function, but if it can, refactor the code to handle that case.

24. Incorrect dataPricer model update in ProgramKeepalive, causing lower cost and demand

Severity: **High**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-24
Target: `stylus/arbos/programs/programs.go`	

Description

When the `ProgramKeepalive` function calls the `dataPricer.UpdateModel` function, it passes in the number of program bytes in kilobytes instead of in bytes (figures 24.1–24.2). As a result, the computed demand and cost values in wei are lower than intended (figure 24.3).

```
func (p Programs) ProgramKeepalive(codeHash common.Hash, time uint64) (*big.Int,
error) {
	program, err := p.getProgram(codeHash, time)
	(...)
	cost, err := p.dataPricer.UpdateModel(program.asmEstimate.ToUint32(), time)
```

Figure 24.1: `stylus/arbos/programs/programs.go#L429–L450`

```
type Program struct {
	version     uint16
	initGas     uint24
	asmEstimate uint24 // Unit is a kb (predicted canonically)
	(...)
```

Figure 24.2: `stylus/arbos/programs/programs.go#L40–L47`

```
func (p *DataPricer) UpdateModel(tempBytes uint32, time uint64) (*big.Int, error) {
	demand, _ := p.demand.Get()
	(...)
	demand = arbmath.SaturatingUSub(demand, credit)
	demand = arbmath.SaturatingUAdd(demand, tempBytes)

	if err := p.demand.Set(demand); err != nil {
		return nil, err
	}
	(...)
	costInWei := arbmath.SaturatingUMul(costPerByte, uint64(tempBytes))
	return arbmath.UintToBig(costInWei), nil
}
```

Figure 24.3: `stylus/arbos/programs/data_pricer.go#L61–L88`

Note that when a program is activated, the `DataPricer.UpdateModel` is called correctly with the number of program bytes instead of kilobytes (`stylus/arbos/programs/programs.go#L246–L263`). This is because it is called with the `info.asmEstimate` variable (from the `activationInfo.asmEstimate` field), which is in bytes, instead of the `estimateKb` variable, which is in kilobytes and which is saved into the `Program.asmEstimate` field.

Exploit Scenario
A chain owner sets a keepalive for a program, resulting in an incorrect data price model update and a cheaper execution of the keepalive function.

Recommendations
Short term, take the following actions:

- Fix the `ProgramKeepalive` function so that it passes in the number of program bytes in bytes instead of kilobytes to the `dataPrice.UpdateModel` function. Note that this may require code changes in the `ActivateProgram` function as well so that both price model update calls receive the same value for the program bytes amount.
- Change the name of the `asmEstimate` field in the `Program` type to `asmEstimateKb` to prevent similar issues in the future (unless the field is refactored to hold the number of bytes).

Long term, add tests for this functionality.

25. Machine does not properly handle WASM binaries with both Rust and Go support

Severity: **Low**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-STYLUS-25
Target: `stylus/arbitrator/prover/src/machine.rs`	

Description

The `from_binaries` function parses WASM modules from binaries that have either Rust or Go support; however, the function may detect both a Rust and Go binary at the same time (figure 25.1). This would cause an incorrect entrypoint code to be generated from both the Rust and Go support additions.

A user could create a module that triggers both Rust and Go support by creating a function named `run` using the `no_mangle` attribute in a Rust program and compiling it to a WASM module.

```
pub fn from_binaries(/* ... */) -> Result<Machine> {
       // Rust support
        let rust_fn = "__main_void";
        if let Some(&f) = main_exports.get(rust_fn).filter(|_| runtime_support) {
            let expected_type = FunctionType::new([], [I32]);
            ensure!(
                main_module.func_types[f as usize] == expected_type,
                "Main function doesn't match expected signature of [] -> [ret]",
            );
            entry!(@cross, u32::try_from(main_module_idx).unwrap(), f);
            entry!(Drop);
            entry!(HaltAndSetFinished);
        }

        // Go support
        if let Some(&f) = main_exports.get("run").filter(|_| runtime_support) {
            let mut expected_type = FunctionType::default();
            (...)
           // Launch main with an argument count of 1 and argv_ptr
            entry!(I32Const, 1);
            entry!(I32Const, argv_ptr);
            entry!(@cross, main_module_idx, f);
            (...)
        }
```

Figure 25.1: `stylus/arbitrator/prover/src/machine.rs#L1194–L1260`

Exploit Scenario

A user creates a Rust program that includes a `run` function marked with the `no_mangle` attribute and compiles it to a WASM module to deploy it to the network. The user wastes funds deploying and activating the module, as it ends up being unusable due to the creation of incorrect entrypoint code during the WASM module parsing process.

Recommendations

Short term, have the `from_binaries` function check whether both Rust and Go support is included and, if so, error out the processing and inform the user that they cannot have both function names. Additionally, have the function log a message to inform the user whenever Rust or Go support is detected and that the entrypoint code has been instrumented as such. This will help users to understand how their code has been instrumented.

26. Computation of internal stack hash uses wrong prefix string	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-STYLUS-26
Target: `stylus/arbitrator/prover/src/machine.rs`	

Description

The `prover::machine::Machine::stack_hashes` function computes hashes of the co-thread frame stacks, value stacks, and internal stack using a prefix string (figure 26.1). The value stack and the internal stack pass in the same prefix ("`Value`") to the hash computation macros, so certain sub-hashes of the value stack (`first_hash`, shown in the figure, and `last_hash`, omitted from the figure) may have the same value as the internal stack hash.

This does not seem to create any security risk, but it seems that the prefix for the internal stack was intended to be different from other stack prefixes.

```
fn stack_hashes(&self) -> (FrameStackHash, ValueStackHash, InterStackHash) {
    macro_rules! compute {
        ($stack:expr, $prefix:expr) => {{
            let frames = $stack.iter().map(|v| v.hash());
            hash_stack(frames, concat!($prefix, " stack:"))
        }};
    }
    macro_rules! compute_multistack {
        ($field:expr, $stacks:expr, $prefix:expr, $hasher: expr) => {{
            let first_elem = *$stacks.first().unwrap();
            let first_hash = hash_stack(
                first_elem.iter().map(|v| v.hash()),
                concat!($prefix, " stack:"),
            );
            // (...) - more code
        }};
    }
    let frame_stacks = compute_multistack!(/* (...) */, "Stack frame",/* (...) */);
    let value_stacks = compute_multistack!(/* (...) */, "Value", /* (...) */);
    let inter_stack = compute!(self.internal_stack, "Value");

    (frame_stacks, value_stacks, inter_stack)
}
```

Figure 26.1: `stylus/arbitrator/prover/src/machine.rs#L2703-L2767`

Recommendations

Change the prefix used for the internal stack hash computation in the `stack_hashes` function to `“Internal”`. While this may not change any security property of the system, it will remove a possibility of a hash collision (between the internal stack hash and a partial hash from the value stack), which could create confusion if seen.

27. WASI preview 1 may be incompatible with future versions	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-STYLUS-27
Target: Stylus interop layer, WASI	

Description

Stylus was recently updated to use Go 1.21's WASI preview 1 for its WASM execution. (Previously, WASM was run through a JavaScript engine embedded in the Rust code.) However, since this iteration of WASI is only a preview and, according to a related issue on Go's GitHub repository, "this interface is evolving without the insurance of backward compatibility," it may require additional effort to add support for WASI preview 2 and future WASI versions.

Recommendations

Long term, track the developments of support for WASI preview 2 in Go. Make sure to work around any version incompatibilities when updating the Stylus codebase to future WASI versions.

References

- WASI preview 2 meeting presentation (June 2022)
- `golang/go#65333`: Go issue tracking WASI preview 2 support

28. Possible out-of-bounds write in strncpy function in Stylus C SDK

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-STYLUS-28
Target: `stylus/arbitrator/langs/c/src/simplelib.c`	

Description

The `strncpy` function defined in the Stylus C SDK writes past the destination string when the source string (`src`) is shorter than the number of bytes (`num`) to write to the destination string (figure 28.1).

This causes another area of the memory of the program to be overwritten, which may have various consequences depending on the program code and its memory layout.

```
char *strncpy(char *dst, const char *src, size_t num) {
    size_t idx=0;
    while (idx<num && src[idx]!=0) {
        idx++;
    }
    memcpy(dst, src, idx);
    if (idx < num) {
        memset(dst+num, 0, num-idx);
    }
    return dst;
}
```

Figure 28.1: `stylus/arbitrator/langs/c/src/simplelib.c#L6–L16`

This bug can be detected by compiling an example program using this function (figure 28.2) with ASan (by using the `-fsanitize=address` flag) with the GCC or Clang compiler.

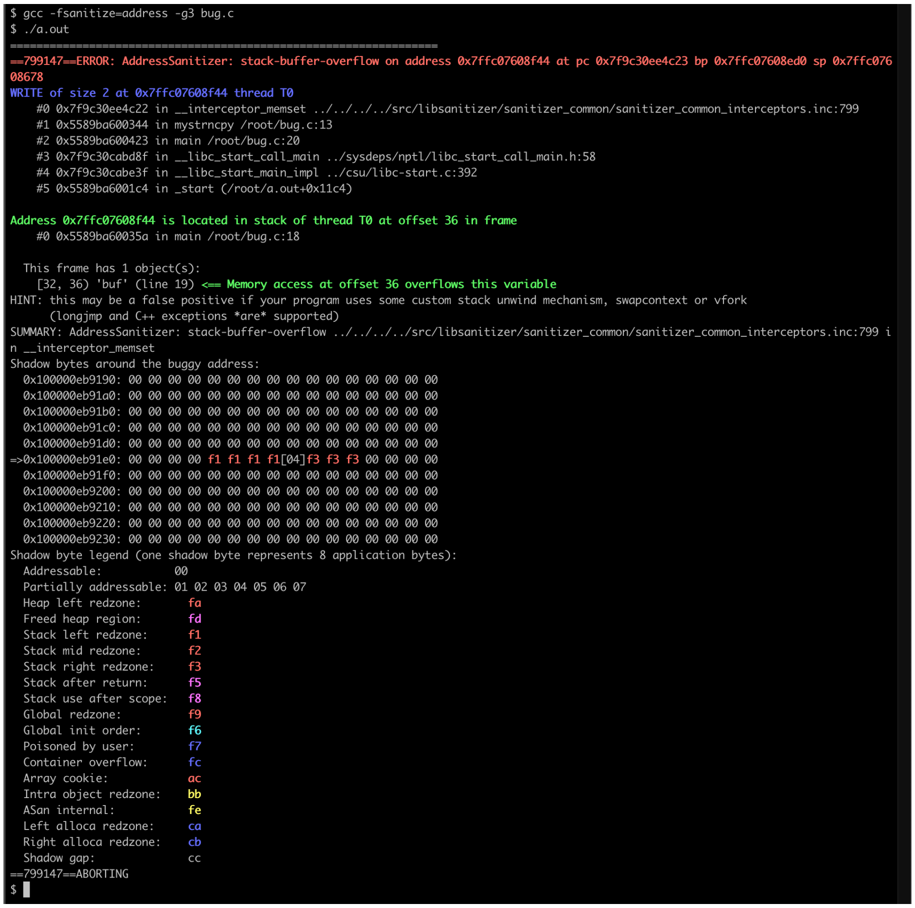
```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

char *mystrncpy(char *dst, const char *src, size_t num) {
    // code from Figure 28.1
}

int main() {
       char buf[4] = {0};
       mystrncpy(buf, "ab", 4);
       printf("buf='%s'\n", buf);
```

```
}
```

Figure 28.2: An example program that triggers the bug described in the finding



```
$ gcc -fsanitize=address -g3 bug.c
$ ./a.out
=================================================================
==799147==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffc07608f44 at pc 0x7f9c30ee4c23 bp 0x7ffc07608ed0 sp 0x7ffc076
08678
WRITE of size 2 at 0x7ffc07608f44 thread T0
    #0 0x7f9c30ee4c22 in __interceptor_memset ../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:799
    #1 0x5589ba600344 in mystrncpy /root/bug.c:13
    #2 0x5589ba600423 in main /root/bug.c:20
    #3 0x7f9c30cabd8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #4 0x7f9c30cabe3f in __libc_start_main_impl ../csu/libc-start.c:392
    #5 0x5589ba6001c4 in _start (/root/a.out+0x11c4)

Address 0x7ffc07608f44 is located in stack of thread T0 at offset 36 in frame
    #0 0x5589ba60035a in main /root/bug.c:18

  This frame has 1 object(s):
    [32, 36) 'buf' (line 19) <== Memory access at offset 36 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow ../../../../src/libsanitizer/sanitizer_common/sanitizer_common_interceptors.inc:799 i
n __interceptor_memset
Shadow bytes around the buggy address:
  0x100000eb9190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000eb91a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000eb91b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000eb91c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000eb91d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x100000eb91e0: 00 00 00 00 f1 f1 f1 f1[04]f3 f3 f3 00 00 00 00
  0x100000eb91f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000eb9200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000eb9210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000eb9220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x100000eb9230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==799147==ABORTING
$
```

Figure 28.3: Output from the example program, showing that it detects this issue

Recommendations

Short term, change the problematic line to `memset(dst+idx, 0, num-idx);` to prevent the issue described in this finding.

Long term, implement tests for edge-case inputs for the Stylus SDK functions.

References

- `strncpy` manual page

29. Insufficient out-of-bounds check in memcpy utility function for ConstString

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-STYLUS-29
Target: `stylus/arbitrator/langs/rust/stylus-sdk/src/abi/const_string.rs`	

Description

The `memcpy` utility function, used to implement `ConstString` functions in the Stylus Rust SDK, contains an insufficient check against out-of-bounds conditions: it misses the following conditions that would cause a program to write past the destination buffer:

- The offset is equal to the destination length.
- The source length is larger than the destination length.

```
/// Copies data from `source` to `dest` in a `const` context.
/// This function is very inefficient for other purposes.
const fn memcpy<const N: usize>(
    mut source: &[u8],
    mut dest: [u8; N],
    mut offset: usize,
) -> [u8; N] {
    if offset > dest.len() {
        panic!("out-of-bounds memcpy");
    }
    while !source.is_empty() {
        dest[offset] = source[0];
        offset += 1;
        (_, source) = source.split_at(1);
    }
    dest
}
```

Figure 29.1: *`stylus/arbitrator/langs/rust/stylus-sdk/src/abi/const_string.rs#L26–L40`*

Recommendations

Short term, change the insufficient out-of-bounds check in the `memcpy` function to `if offset + source.len() >= dest.len()` to prevent potential bugs that could occur if the function were used incorrectly.

Long term, implement tests for edge case inputs for the Stylus SDK functions.

30. Unused and unset timeouts in Arbitrator's JIT code

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-STYLUS-30
Target: Arbitrator JIT code	

Description

There are potential issues with timeouts in the Arbitrator's JIT code:

1. Read and write operations for sockets created in the `ready_hostio` function (figure 30.1) have no timeouts. If the server the Arbitrator connects to does not send any data, the lack of timeout could result in a denial of service.
2. The `ProcessEnv::child_timeout` field, which is set to 15 seconds (figure 30.2), is unused across the codebase.

```
fn ready_hostio(env: &mut WasmEnv) -> MaybeEscape {
    {...omitted for brevity...}
    let socket = TcpStream::connect(&address)?;
    socket.set_nodelay(true)?;
    // no call to socket.set_{read,write}_timeout

    let mut reader = BufReader::new(socket.try_clone()?);
```

Figure 30.1: `stylus/arbitrator/jit/src/wavmio.rs#L198-L303`

```
impl Default for ProcessEnv {
    fn default() -> Self {
        Self {
            forks: false,
            debug: false,
            socket: None,
            last_preimage: None,
            timestamp: Instant::now(),
            child_timeout: Duration::from_secs(15),
            reached_wavmio: false,
        }
    }
}
```

Figure 30.2: `stylus/arbitrator/jit/src/machine.rs#L331-L342`

Recommendations

Short term, take the following actions:

- Set timeouts for read and write operations for sockets created in the `ready_hostio` function.
- Remove the `Process::child_timeout` field or refactor the code to use it.

31. New machine hashing format breaks backward compatibility

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-STYLUS-31
Target: `stylus/contracts/src/state/Machine.sol`	

Description

The new hashing format of the One Step Proof (OSP) contracts for the Stylus VM includes new hashing fields that break backward compatibility for the Nitro VM.

The machine hash of the OSP contracts captures the entirety of the Stylus VM's state.

```
function hash(Machine memory mach) internal pure returns (bytes32) {
    // Warning: the non-running hashes are replicated in Challenge
    if (mach.status == MachineStatus.RUNNING) {
        bytes32 valueMultiHash = mach.valueMultiStack.hash(
            mach.valueStack.hash(),
            mach.recoveryPc != NO_RECOVERY_PC
        );
        bytes32 frameMultiHash = mach.frameMultiStack.hash(
            mach.frameStack.hash(),
            mach.recoveryPc != NO_RECOVERY_PC
        );
        bytes memory preimage = abi.encodePacked(
            "Machine running:",
            valueMultiHash,
            mach.internalStack.hash(),
            frameMultiHash,
            mach.globalStateHash,
            mach.moduleIdx,
            mach.functionIdx,
            mach.functionPc,
            mach.recoveryPc,
            mach.modulesRoot
        );
        return keccak256(preimage);
    } else if (mach.status == MachineStatus.FINISHED) {
        return keccak256(abi.encodePacked("Machine finished:",
mach.globalStateHash));
    } else if (mach.status == MachineStatus.ERRORED) {
        return keccak256(abi.encodePacked("Machine errored:"));
    } else if (mach.status == MachineStatus.TOO_FAR) {
        return keccak256(abi.encodePacked("Machine too far:"));
    } else {
        revert("BAD_MACH_STATUS");
```

```
    }
}
```

Figure 31.1: The function that creates the hash for the Stylus VM (stylus-contracts/src/state/Machine.sol#41–74)

The hashing format of the Stylus VM has been updated from the format used to hash the Nitro VM, shown in figure 31.2; the new format includes multistacks (stacks of stacks) and a recovery program counter.

```
function hash(Machine memory mach) internal pure returns (bytes32) {
    // Warning: the non-running hashes are replicated in Challenge
    if (mach.status == MachineStatus.RUNNING) {
        return
            keccak256(
                abi.encodePacked(
                    "Machine running:",
                    mach.valueStack.hash(),
                    mach.internalStack.hash(),
                    mach.frameStack.hash(),
                    mach.globalStateHash,
                    mach.moduleIdx,
                    mach.functionIdx,
                    mach.functionPc,
                    mach.modulesRoot
                )
            );
    } else if (mach.status == MachineStatus.FINISHED) {
        return keccak256(abi.encodePacked("Machine finished:",
mach.globalStateHash));
    } else if (mach.status == MachineStatus.ERRORED) {
        return keccak256(abi.encodePacked("Machine errored:"));
    } else if (mach.status == MachineStatus.TOO_FAR) {
        return keccak256(abi.encodePacked("Machine too far:"));
    } else {
        revert("BAD_MACH_STATUS");
    }
}
```

Figure 31.2: The function that creates the hash for the Nitro VM (https://etherscan.io/address/0x3E1f62AA8076000c3218493FE3e0Ae40bcB9A1DF#code)

The discrepancy means that the Stylus VM upgrade will cause an inconsistent state between the hash of the Stylus VM and the previous Nitro VM hash, which is important to take into account when fraud proving is activated.

Exploit Scenario

Alice and Bob enter a challenge before the upgrade of the Stylus VM and OSP contracts. The upgrade occurs and causes a mismatch between the current and previous machine states, so the OSP cannot be run and Alice and Bob are both blocked from proving their state. Bob loses the challenge due to a timeout.

Recommendations

Short term, ensure that the fraud proving system is deactivated during the Stylus VM upgrade.

Long term, thoroughly document the risks associated with breaking backward compatibility of the machine hash and whether/how the network's normal operation can be affected during an upgrade.

32. Unclear handling of unexpected machine state transitions	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-32
Target: `stylus/contracts/src/state/Machine.sol`	

Description

The OSP `Machine` contract does not handle unexpected state transitions when executing a single opcode in a consistent manner.

In some cases (such as when `setPc` is called), the machine enters an errored state when an unexpected value type is found or when the program counter content contains unexpected data.

```
function setPc(Machine memory mach, Value memory pc) internal pure {
    if (pc.valueType == ValueType.REF_NULL) {
        mach.status = MachineStatus.ERRORED;
        return;
    }
    if (pc.valueType != ValueType.INTERNAL_REF) {
        mach.status = MachineStatus.ERRORED;
        return;
    }
    if (!setPcFromData(mach, pc.contents)) {
        mach.status = MachineStatus.ERRORED;
        return;
    }
}
```

Figure 32.1: Unexpected data in the program counter leads to an errored state.
(`stylus-contracts/src/state/Machine.sol#124–137`)

The internal `setPcFromData` function enters an early return condition and does not update the machine state when unexpected data is present.

```
function setPcFromData(Machine memory mach, uint256 data) internal pure returns
(bool) {
    if (data >> 96 != 0) {
        return false;
    }

    mach.functionPc = uint32(data);
    mach.functionIdx = uint32(data >> 32);
    mach.moduleIdx = uint32(data >> 64);
```

```
    return true;
}
```

Figure 32.2: The internal `setPcFromData` function (`stylus-contracts/src/state/Machine.sol#92–101`)

In other cases (such as when the machine is recovering from an errored state and `setPcFromRecovery` fails), this unexpected case is simply ignored.

```
if (mach.status == MachineStatus.ERRORED && mach.recoveryPc !=
MachineLib.NO_RECOVERY_PC) {
    // capture error, recover into main thread.
    mach.switchCoThreadStacks();
    mach.setPcFromRecovery();
    mach.status = MachineStatus.RUNNING;
}
```

Figure 32.3: A failure in setting the program counter is ignored in `mach.setPcFromRecovery`. (`stylus-contracts/src/osp/OneStepProofEntry.sol#135–140`)

```
function setPcFromRecovery(Machine memory mach) internal pure returns (bool) {
    if (!setPcFromData(mach, uint256(mach.recoveryPc))) {
        return false;
    }
    mach.recoveryPc = NO_RECOVERY_PC;
    return true;
}
```

Figure 32.4: The internal `setPcFromRecovery` function returns a Boolean value indicating an unexpected state. (`stylus-contracts/src/state/Machine.sol#103–109`)

In other cases (such as when `assumeI32` is called in `executeCrossModuleInternalCall`), the unexpected value is handled through a `require` check, which essentially blocks the execution of the OSP.

```
function executeCrossModuleInternalCall(
    Machine memory mach,
    Module memory mod,
    Instruction calldata inst,
    bytes calldata proof
) internal pure {
    // Get the target from the stack
    uint32 internalIndex = uint32(inst.argumentData);
    uint32 moduleIndex = mach.valueStack.pop().assumeI32();
```

Figure 32.5: An unexpected state transition cannot be executed. (`stylus-contracts/src/osp/OneStepProver0.sol#167–175`)

```
function assumeI32(Value memory val) internal pure returns (uint32) {
```

```
    uint256 uintval = uint256(val.contents);
    require(val.valueType == ValueType.I32, "NOT_I32");
    require(uintval < (1 << 32), "BAD_I32");
    return uint32(uintval);
}
```

Figure 32.6: The `assumeI32` function requires the value to be of the expected data format and blocks execution otherwise. (`stylus-contracts/src/state/Value.sol#31–36`)

In order to have a clearly defined incident response plan, unexpected state transitions should be handled consistently.

Recommendations

Short term, have the machine handle all listed unexpected machine state transitions from the OSP in the same way (e.g., by transitioning into an errored state).

Long term, document all the invalid state transitions across components and decide on a sound and safe strategy to handle them.

33. Potential footguns and attack vectors due to new memory model	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-33
Target: `arbos/programs/programs.go`	

Description

The Stylus memory model introduces new concepts that might be surprising to developers who are familiar with the EVM model; these new concepts could also introduce potential attack vectors.

The Stylus memory model uses a global memory model, in which each new memory page allocation is priced exponentially given the number of pages shared across all user programs. This is in contrast to the EVM, which prices memory quadratically and independently of other programs'/contracts' use of memory.

With certain patterns (e.g., ERC-4337 `UserOperation` forwarding/relaying), it may be essential to have predictable costs for memory expansion in the current context in order to ensure that relayed calls are executed with the conditions the original signer intended. Because a relayed call typically involves handling memory, these costs must be taken into account for the outer call that wraps the inner call. If these costs can be influenced by previous user programs allocating a large number of memory pages, it might open up new attack vectors.

Exploit Scenario

A relaying contract wraps an inner call with a fixed amount of gas. The inner call requires memory allocation. Because the outer call can open an arbitrary number of memory pages, the inner call fails unexpectedly due to the increased gas cost of global memory allocation.

Recommendations

Long term, make developers aware of any deviation from the EVM model and its potential security considerations.

34. Storage cache can become out of sync for reentrant and delegated calls

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-STYLUS-34
Target: `stylus/arbitrator/stylus/src/lib.rs`, `stylus/arbitrator/arbutil/src/evm/req.rs`	

Description

A storage cache's known values can become out of sync, causing storage reads to be outdated and storage write operations to be omitted.

Storage caches take into account only their current call context. Every Stylus program call creates a new EVM API requestor (`EvmApiRequestor`).

```
#[no_mangle]
pub unsafe extern "C" fn stylus_call(
    module: GoSliceData,
    calldata: GoSliceData,
    config: StylusConfig,
    req_handler: NativeRequestHandler,
    evm_data: EvmData,
    debug_chain: u32,
    output: *mut RustBytes,
    gas: *mut u64,
) -> UserOutcomeKind {
    let module = module.slice();
    let calldata = calldata.slice().to_vec();
    let compile = CompileConfig::version(config.version, debug_chain != 0);
    let evm_api = EvmApiRequestor::new(req_handler);
    let pricing = config.pricing;
    let output = &mut *output;
    let ink = pricing.gas_to_ink(*gas);

    // ...
}
```

Figure 34.1: A call to a Stylus program creates a new EVM API requestor.
(`stylus/arbitrator/stylus/src/lib.rs#169–205`)

When a new EVM API requestor is created, a new `StorageCache` struct is created as well.

```
impl<D: DataReader, H: RequestHandler<D>> EvmApiRequestor<D, H> {
    pub fn new(handler: H) -> Self {
        Self {
```

```
            handler,
            last_code: None,
            last_return_data: None,
            storage_cache: StorageCache::default(),
        }
    }
```

Figure 34.2: A new storage cache is created.
(stylus/arbitrator/arbutil/src/evm/req.rs#28–36)

When there is no need to share storage state between two calls, storage caches can operate independently of each other without any issues. However, in the EVM contract, storage state is shared for delegated and reentrant calls.

A call that would share storage state would also create a new storage cache struct, which can cause the first storage cache to become out of sync when the second cache modifies some of the first cache's "known" values. Known values are those that the storage cache thinks are located in the state trie.

Such situations could cause storage reads to be incorrect or outdated and write operations to be omitted.

Exploit Scenario

A multisignature Stylus program `SmartWallet` allows arbitrary program execution with one important invariant: the ownership of the program is not allowed to change after the execution of the inner call (figure 34.3). Because the inner call is a reentrant call, the storage cache becomes out of sync; this causes the ownership invariant check to be faulty, allowing it to be bypassed (figures 34.4–34.5).

```
#![no_main]

use stylus_sdk::{
    alloy_primitives::Address,
    call::RawCall,
    console,
    stylus_proc::{entrypoint, external, sol_storage},
};

extern crate alloc;

#[global_allocator]
static ALLOC: mini_alloc::MiniAlloc = mini_alloc::MiniAlloc::INIT;

sol_storage! {
    #[entrypoint]
    pub struct SmartWallet {
        address owner;
        bool initialized;
    }
```

```rust
}

#[external]
impl SmartWallet {
    pub fn owner(&self) -> Result<Address, String> {
        Ok(self.owner.get())
    }

    pub fn initialize(&mut self, owner: Address) -> Result<(), String> {
        if self.initialized.get() {
            return Err("Already initialized".into());
        }

        self.owner.set(owner);
        self.initialized.set(true);

        Ok(())
    }

    pub fn execute(&mut self, args: Vec<u8>) -> Result<Vec<u8>, Vec<u8>> {
        // ... some multisig access controls
        let previous_owner = self.owner.get();

        let mut args = &args[..];

        let mut take_args = |n_bytes: usize| -> &[u8] {
            let value = &args[..n_bytes];
            args = &args[n_bytes..];
            value
        };

        let kind = take_args(1)[0];
        let addr = Address::try_from(take_args(20)).unwrap();
        let raw_call = match kind {
            0 => RawCall::new(),
            1 => RawCall::new_delegate(),
            2 => RawCall::new_static(),
            x => panic!("unknown call kind {x}"),
        };

        let return_data = raw_call.call(addr, args)?;

        assert_eq!(
            previous_owner,
            self.owner.get(),
            "Owner cannot change during `execute` call"
        );

        Ok(return_data)
    }
}
```

Figure 34.3: A multisignature wallet that includes an invariant that the program ownership must not change after the execution of the inner call

```
func TestProgramSmartWalletPoc(t *testing.T) {
	t.Parallel()
	testSmartWalletPoc(t, true)
}

func testSmartWalletPoc(t *testing.T, jit bool) {
	builder, auth, cleanup := setupProgramTest(t, jit)
	ctx := builder.ctx
	l2info := builder.L2Info
	l2client := builder.L2.Client
	defer cleanup()

	ownerAddress := l2info.GetAddress("Owner")

	programAddr := deployWasm(t, ctx, auth, l2client,
"../arbitrator/stylus/tests/storage-poc/target/wasm32-unknown-unknown/release/storag
e-poc.wasm")
	storageAddr := deployWasm(t, ctx, auth, l2client, rustFile("storage"))

	colors.PrintGrey("storage.wasm   ", storageAddr)
	colors.PrintGrey("storage-poc.wasm ", programAddr)

	programsAbi :=
`[{"type":"function","name":"execute","inputs":[{"name":"args","type":"uint8[]"}],"o
utputs":[],"stateMutability":"nonpayable"},{"type":"function","name":"initialize","i
nputs":[{"name":"owner","type":"address","internalType":"address"}],"outputs":[],"st
ateMutability":"nonpayable"},{"type":"function","name":"owner","inputs":[],"outputs"
:[{"name":"","type":"uint256","internalType":"uint256"}],"stateMutability":"view"}]`

	callOwner, _ := util.NewCallParser(programsAbi, "owner")
	callInitialize, _ := util.NewCallParser(programsAbi, "initialize")
	callExecute, _ := util.NewCallParser(programsAbi, "execute")

	ensure := func(tx *types.Transaction, err error) *types.Receipt {
		t.Helper()
		Require(t, err)
		receipt, err := EnsureTxSucceeded(ctx, l2client, tx)
		Require(t, err)
		return receipt
	}
	pack := func(data []byte, err error) []byte {
		Require(t, err)
		return data
	}
	assertProgramOwnership := func() {

		args, _ := callOwner()
		returnData := sendContractCall(t, ctx, programAddr, l2client, args)
		newOwner := common.BytesToAddress(returnData)
		if ownerAddress == newOwner {
```

```go
				colors.PrintRed("Ownership remains")
			} else {
				Fatal(t, "Owner changed", ownerAddress, newOwner)
			}
		}

		tx := l2info.PrepareTxTo("Owner", &programAddr, 1e9, nil,
pack(callInitialize(ownerAddress)))
		ensure(tx, l2client.SendTransaction(ctx, tx))

		// "Owner" remains the owner of the program.
		assertProgramOwnership()

		key := common.Hash{}
		value := common.HexToHash("0xdead")

		args := []uint8{}
		args = append(args, 0x01)                    // delegatecall
		args = append(args, storageAddr.Bytes()...) // storage address
		args = append(args, 0x01)                    // storage write op
		args = append(args, key.Bytes()...)          // key
		args = append(args, value.Bytes()...)        // value

		tx = l2info.PrepareTxTo("Owner", &programAddr, 1e9, nil,
pack(callExecute(args)))
		ensure(tx, l2client.SendTransaction(ctx, tx))

		// This passes
		// The `owner` address has been modified through the call to `execute`.
		assertStorageAt(t, ctx, l2client, programAddr, key, value)
		// This fails
		// "Owner" is not the owner of the program anymore.
		assertProgramOwnership()

		validateBlocks(t, 1, jit, builder)
}
```

Figure 34.4: The Go system test, which is able to bypass `SmartWallet`'s ownership invariant

```
go test ./system_tests/...  -run ^TestProgramSmartWalletPoc$
...
Ownership remains
...
--- FAIL: TestProgramSmartWalletPoc (0.81s)
    program_test.go:1096:  [Owner changed 0x26E554a8acF9003b83495c7f45F06edCB803d4e3
0x000000000000000000000000000000000000dEaD]
FAIL
FAIL	github.com/offchainlabs/nitro/system_tests	1.735s
FAIL
```

Figure 34.5: The program ownership is changed.

Recommendations

Short term, modify the associated code so that the storage cache's values are committed beforehand whenever delegated or reentrant calls are possible. Alternatively, consider sharing storage caches between call frames. However, the second option will likely come with significant code inefficiencies and overhead.

Long term, thoroughly document the intended behavior of the cache, including whether it should persist across calls and any potentially unsafe uses for Stylus developers.

35. Storage cache can be written to in a static call context	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-STYLUS-35
Target: `stylus/arbitrator/arbutil/src/evm/req.rs`	

Description

The storage cache can be written to inside of a static call context, which can lead to confusing and unexpected behavior.

The storage cache is intended to minimize storage read and write operations. When the storage cache is flushed, only the values that have changed from the known values (i.e., values that are "dirty") are committed to the persistent storage state, via the `EvmApiMethod::SetTrieSlots` method.

```
fn flush_storage_cache(&mut self, clear: bool, gas_left: u64) -> Result<u64> {
    let mut data = Vec::with_capacity(64 * self.storage_cache.len() + 8);
    data.extend(gas_left.to_be_bytes());

    for (key, value) in &mut self.storage_cache.slots {
        if value.dirty() {
            data.extend(*key);
            data.extend(*value.value);
            value.known = Some(value.value);
        }
    }
    if clear {
        self.storage_cache.clear();
    }
    if data.len() == 8 {
        return Ok(0); // no need to make request
    }

    let (res, _, cost) = self.request(EvmApiMethod::SetTrieSlots, data);
    if res[0] != EvmApiStatus::Success.into() {
        bail!("{}", String::from_utf8_or_hex(res));
    }
    Ok(cost)
}
```

Figure 35.1: Only dirty values are committed to persistent state when the storage cache is flushed. (`stylus/arbitrator/arbutil/src/evm/req.rs#122–145`)

Values that are not dirty do not result in `EvmApiMethod::SetTrieSlots` requests.

In order for a value to be known, it must be either retrieved from Geth via the `GetBytes32` EVM API method or committed by the storage cache itself via the `SetTrieSlots` EVM API method.

This means that a get request can change the behavior of a subsequent storage cache flush host I/O operation, leading to strange and unexpected behavior inside a static call context where persistent state changes are not permitted.

Exploit Scenario

Inside of a static call context, storage writes are not allowed. However, writing multiple values to the storage cache is allowed if they end up equaling the known values.

```
#![no_main]

use stylus_sdk::{
    alloy_primitives::{B256, U256},
    call::RawCall,
    console, contract, msg,
    storage::{GlobalStorage, StorageCache},
    stylus_proc::entrypoint,
};

extern crate alloc;

#[global_allocator]
static ALLOC: mini_alloc::MiniAlloc = mini_alloc::MiniAlloc::INIT;

#[entrypoint]
fn user_main(_input: Vec<u8>) -> Result<Vec<u8>, Vec<u8>> {
    let slot = U256::from(0);

    let get = |slot| {
        let value = StorageCache::get_word(slot);
        console!("StorageCache::get_word({slot}) -> {value}");
    };
    let set = |slot, value| {
        console!("StorageCache::set_word({slot}, {value})");
        unsafe { StorageCache::set_word(slot, value) };
    };
    let flush = || {
        console!("StorageCache::flush()");
        StorageCache::flush();
    };
    if msg::reentrant() {
        get(slot); // If this line is removed, the staticcall fails.

        // Inside staticcall context.
        set(slot, B256::new([0xaa; 32]));
        set(slot, B256::new([0xbb; 32]));
        set(slot, B256::new([0x00; 32]));
```

```
        flush();
    } else {
        // Make reentrant static call.
        let address = contract::address();
        unsafe { RawCall::new_static().call(address, &[])? };
    }

    Ok(vec![])
}
```

Figure 35.2: The static call fails if a previous `GetBytes32` EVM API request is removed.

Recommendations

Short term, consider forbidding writes to the storage cache inside of a static call context. This is especially important if the storage cache is to be shared among reentrant calls, as explained in the issue TOB-STYLUS-34, as a static call should not be able to influence another call's behavior through shared state (aside from gas costs).

Long term, be aware of optimizations that could lead to strange and confusing patterns when interacting with the system on a higher level.

36. Revert conditions always override user returned status

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-STYLUS-36
Target: `arbitrator/wasm-libraries/user-host/src/link.rs`	

Description

Certain corner conditions in Stylus program execution can cause valid executions to be flagged as reverts.

Once a Stylus program exits early, the `early_exit` flag is used to indicate that `early` should be set as the exit code in the `program_internal__set_done` function (figure 36.1).

```
#[no_mangle]
pub unsafe extern "C" fn program_internal__set_done(mut status: UserOutcomeKind) ->
u32 {
    use UserOutcomeKind::*;

    let program = Program::current();
    let module = program.module;
    let mut outs = program.outs.as_slice();
    let mut ink_left = program_ink_left(module);

    // apply any early exit codes
    if let Some(early) = program.early_exit {
        status = early;
    }

    // check if instrumentation stopped the program
    if program_ink_status(module) != 0 {
        status = OutOfInk;
        outs = &[];
        ink_left = 0;
    }
    if program_stack_left(module) == 0 {
        status = OutOfStack;
        outs = &[];
        ink_left = 0;
    }

    let gas_left = program.config.pricing.ink_to_gas(ink_left);

    let mut output = Vec::with_capacity(8 + outs.len());
    output.extend(gas_left.to_be_bytes());
```

```
    output.extend(outs);
    program
        .request_handler()
        .set_request(status as u32, &output)
}
```

Figure 36.1: The `program_internal__set_done` function in `arbitrator/wasm-libraries/user-host/src/link.rs#L194–L228`

However, this function can override the status returned for program executions if either the ink amount or the stack size is zero, flagging them as reverts. Both of these conditions can be reached if a program exits early.

Exploit Scenario

Alice optimizes a Stylus program execution to use exactly a certain amount of ink in the context of a larger DeFi system executing untrusted calls. Her program is called with the exact amount of ink required to run, so it exits with zero ink left. However, the execution is flagged as a revert.

Recommendations

Short term, consider changing the `program_internal__set_done` function so that valid executions resulting in zero gas are not automatically flagged as reverts, making sure the common out-of-gas and out-of-stack executions are handled correctly.

Long term, review the local and global invariants behind each component to make sure corner cases are correctly defined and handled.

37. CacheManager bids cannot be increased	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-STYLUS-37
Target: `stylus-contracts/src/chain/CacheManager.sol`	

Description

Bids in the cache manager placed on a particular code hash cannot be modified and do not accumulate.

When a bid is placed, the `CacheManager` Solidity contract checks whether the code hash is currently cached and reverts the bid if so.

```
/// Places a bid, reverting if payment is insufficient.
function placeBid(bytes32 codehash) external payable {
    if (isPaused) {
        revert BidsArePaused();
    }
    if (_isCached(codehash)) {
        revert AlreadyCached(codehash);
    }

    uint64 asm = _asmSize(codehash);
    (uint256 bid, uint64 index) = _makeSpace(asm);
    return _addBid(bid, codehash, asm, index);
}
```

Figure 37.1: This check prevents bids from being placed on already cached programs. (`stylus-contracts/src/chain/CacheManager.sol#104–144`)

This makes it impossible to increase a bid before the program is evicted either due to other bids being placed or through sufficient calls to `makeSpace`.

This limitation creates a bad user experience. A user who wants to increase a bid would have to create a new bid, but would first have to pay to evict the program. It might also make it difficult for a popular dapp with many low-capital users to coordinate and combine their funds for a shared bid.

Exploit Scenario

Bob wants to increase a previous bid to his token program. He cannot simply place a new bid; he is required to make sufficient space. He calls `makeSpace` to evict his own program, requiring a 1 ETH payment. In order to add his new 2 ETH bid, he must now pay 3 ETH in total.

Recommendations

Short term, document this limitation of the auction system. Consider adding an alternative unsafe function that does not check whether the code is already cached (however, this would allow multiple entries per code hash). Alternatively, consider adjusting the implementation to allow bids for programs to be increased.

Long term, review the bid mechanisms with user experience in mind; document any sources of friction and ways in which they could be mitigated.

38. The makeSpace function does not refund excess bid value and can be front-run

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-STYLUS-38
Target: `stylus-contracts/src/chain/CacheManager.sol`	

Description

The `makeSpace` function, used to make space for programs in the cache manager, does not refund funds sent above the minimum bid value, even if no state changes are performed.

The `makeSpace` function accepts ETH and requires a minimum bid to be made until enough space is available.

```
/// Evicts entries until enough space exists in the cache, reverting if payment is
insufficient.
/// Returns the new amount of space available on success.
/// Note: will only make up to 5Mb of space. Call repeatedly for more.
function makeSpace(uint64 size) external payable returns (uint64 space) {
    if (size > MAX_MAKE_SPACE) {
        size = MAX_MAKE_SPACE;
    }
    _makeSpace(size);
    return cacheSize - queueSize;
}

/// Evicts entries until enough space exists in the cache, reverting if payment is
insufficient.
/// Returns the bid and the index to use for insertion.
function _makeSpace(uint64 size) internal returns (uint256 bid, uint64 index) {
    // discount historical bids by the number of seconds
    bid = msg.value + block.timestamp * uint256(decay);
    index = uint64(entries.length);

    uint256 min;
    while (queueSize + size > cacheSize) {
        (min, index) = _getBid(bids.pop());
        _deleteEntry(min, index);
    }
    if (bid < min) {
        revert BidTooSmall(bid, min);
    }
}
```

Figure 38.1: The `makeSpace` function requires the minimum bid to be matched until enough space is made. (`stylus-contracts/src/chain/CacheManager.sol#118–144`)

The contract keeps any funds sent above the minimum bid value. This includes the case in which enough space is already available and no funds are required. This can happen, for example, when two calls to `makeSpace` are initiated by different parties.

There is also the possibility that a user calls `makeSpace` to create space, only for that space to be occupied by other bids right after it is freed.

Exploit Scenario

Bob calls `makeSpace` in order to free up space in the cache manager. In the meantime, Alice calls `makeSpace` herself for the same reason. Bob's transaction ends up doing nothing and does not return his funds. Alice is able to insert her program, whereas Bob is where he was at the start.

Recommendations

Short term, have the cache manager refund any excess funds sent above the minimum bid required for making enough space.

Long term, document this behavior so that users are aware of it.

39. Bids do not account for program size	
Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-STYLUS-39
Target: `stylus-contracts/src/chain/CacheManager.sol`	

Description

It is possible for a single bid to evict many programs, regardless of their cumulative price per program byte size, resulting in an unfair auction system.

A program that is to be inserted into the cache manager with a slightly higher bid than many others will be prioritized over those other programs, regardless of the total amount paid per occupied code size. This is because the code for adding a bid for a program does not take into account the program size itself.

```
/// Adds a bid
function _addBid(
    uint256 bid,
    bytes32 code,
    uint64 size,
    uint64 index
) internal {
    if (queueSize + size > cacheSize) {
        revert AsmTooLarge(size, queueSize, cacheSize);
    }

    Entry memory entry = Entry({size: size, code: code});
    ARB_WASM_CACHE.cacheCodehash(code);
    bids.push(_packBid(bid, index));
    queueSize += size;
    if (index == entries.length) {
        entries.push(entry);
    } else {
        entries[index] = entry;
    }
    emit InsertBid(bid, code, size);
}
```

Figure 39.1: The `_addBid` function does not take program size into account (`stylus-contracts/src/chain/CacheManager.sol#145–167`)

Exploit Scenario

There are 50 programs in the cache manager, each of size 0.1 MB and a 1 ETH bid. Bob inserts a new program with a 1.01 ETH bid. If Bob's program size is 0.1 MB, one program will be evicted (1 ETH worth of bids). If the program size is 5 MB, 50 programs will be evicted (50 ETH worth of bids).

Bob's program should not be able to evict any number of programs without paying extra fees.

Recommendations
Short term, consider dividing the bid in `_addBid` by the program size in order to charge a price per byte instead of a fixed price per program.

Long term, thoroughly document the intended behavior of the cache manager in terms of program sizes.

40. Incorrect bid check	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-STYLUS-40
Target: `stylus-contracts/src/chain/CacheManager.sol`	

Description

The `_makeSpace` function allows new bids to go through if they are equal to the current bid (figure 40.1). This is unexpected for an auction system, in which new bids should be considered only if they are superior to previous ones.

```
/// Evicts entries until enough space exists in the cache, reverting if payment is
insufficient.
/// Returns the bid and the index to use for insertion.
function _makeSpace(uint64 size) internal returns (uint192 bid, uint64 index) {
    // discount historical bids by the number of seconds
    bid = uint192(msg.value + block.timestamp * uint256(decay));
    index = uint64(entries.length);

    uint192 min;
    uint64 limit = cacheSize;
    while (queueSize + size > limit) {
        (min, index) = _getBid(bids.pop());
        _deleteEntry(min, index);
    }
    if (bid < min) {
        revert BidTooSmall(bid, min);
    }
}
```

Figure 40.1: The check is a less-than comparison, allowing bids equal to the current bid to be accepted. (`stylus-contracts/src/chain/CacheManager.sol#137–153`)

Recommendations

Short term, replace the check with `bid <= min`.

Long term, thoroughly document the intended behavior of the auction system and use it as a baseline to review its actual behavior.

41. MemoryGrow opcode is underpriced for programs with fixed memory

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-STYLUS-41
Target: `prover/src/programs/meter.rs`, `prover/src/programs/heap.rs`	

Description

The ink charged by the `MemoryGrow` opcode is less than expected for programs that have a fixed memory size.

Stylus defines an ink price for every WASM opcode to be used during program activation. The costs for certain opcodes, such as `MemoryGrow`, are handled by a different part of the code (figure 41.4).

```
pub fn pricing_v1(op: &Operator, tys: &HashMap<SignatureIndex, FunctionType>) -> u64
{
…
let ink = match op {
        {...omitted for brevity...}
        dot!(MemoryGrow) => 1, // cost handled by memory pricer
```

Figure 41.1: Part of the `pricing_v1` function that defers the ink price for `MemoryGrow` to the memory pricer

However, if a WASM program has fixed memory (and therefore does not import the `pay` function), the cost of the opcode will be unmodified (figure 41.2).

```
impl<'a> FuncMiddleware<'a> for FuncHeapBound {
    fn feed<O>(&mut self, op: Operator<'a>, out: &mut O) -> Result<()>
    where
        O: Extend<Operator<'a>>,
    {
        use Operator::*;

        let Some(pay_func) = self.pay_func else {
            out.extend([op]);
            return Ok(());
        };
```

Figure 41.2: The header of the `feed` function of the FuncHeapBound middleware

A call to `MemoryGrow` for a program with a fixed memory returns `-1`, which is correct according to the WASM standard. Unfortunately, the price of that opcode will be 1 ink, which is too small to cover the actual cost of the operation in a WASM execution.

Exploit Scenario

Eve crafts a malicious WASM program that repeatedly triggers the `MemoryGrow` opcode in a WASM program that has a fixed memory in order to exhaust the resources of the validators. Due to the low cost of the `MemoryGrow` opcode on programs with a fixed memory, she pays a minimal amount of ink to carry out the attack.

Recommendations

Short term, increase the cost of the `MemoryGrow` opcode to make sure it is sufficient for all programs, including those with fixed memory.

Long term, perform fuzz testing of the processes for validating, activating, and executing WASM contracts.

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. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

C. Sequence Diagrams

In order to understand the high-level infrastructure and invocations in the code, we used mermaid-js to visualize data validation across the system.

WASM Instrumentation

Figure C.1 illustrates the expected calls of the `config.rs` and `binary.rs` files.

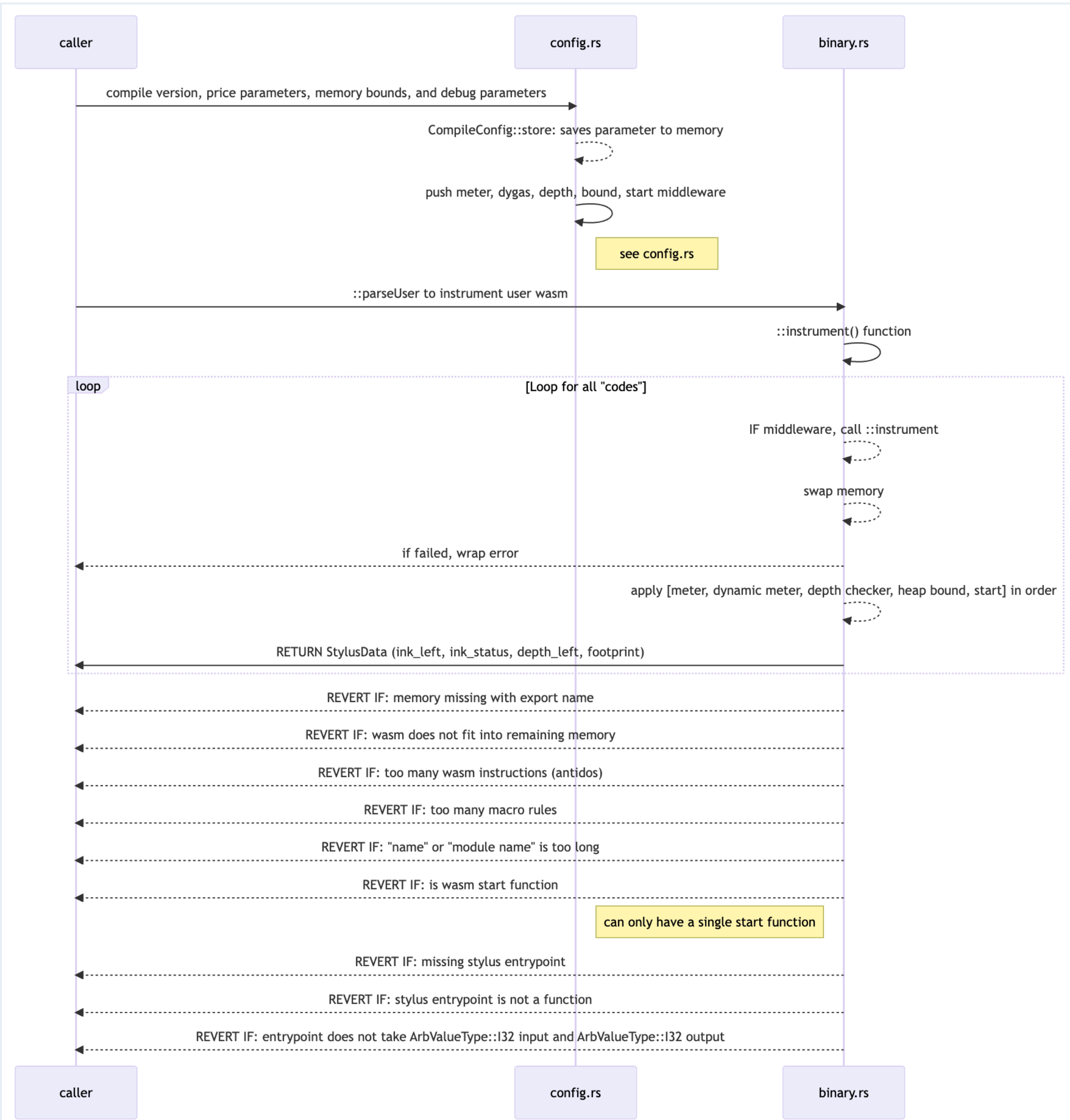


Figure C.1: A sequence diagram of the WASM instrumentation setup

Middleware

Figure C.2 shows the middleware used across the system.

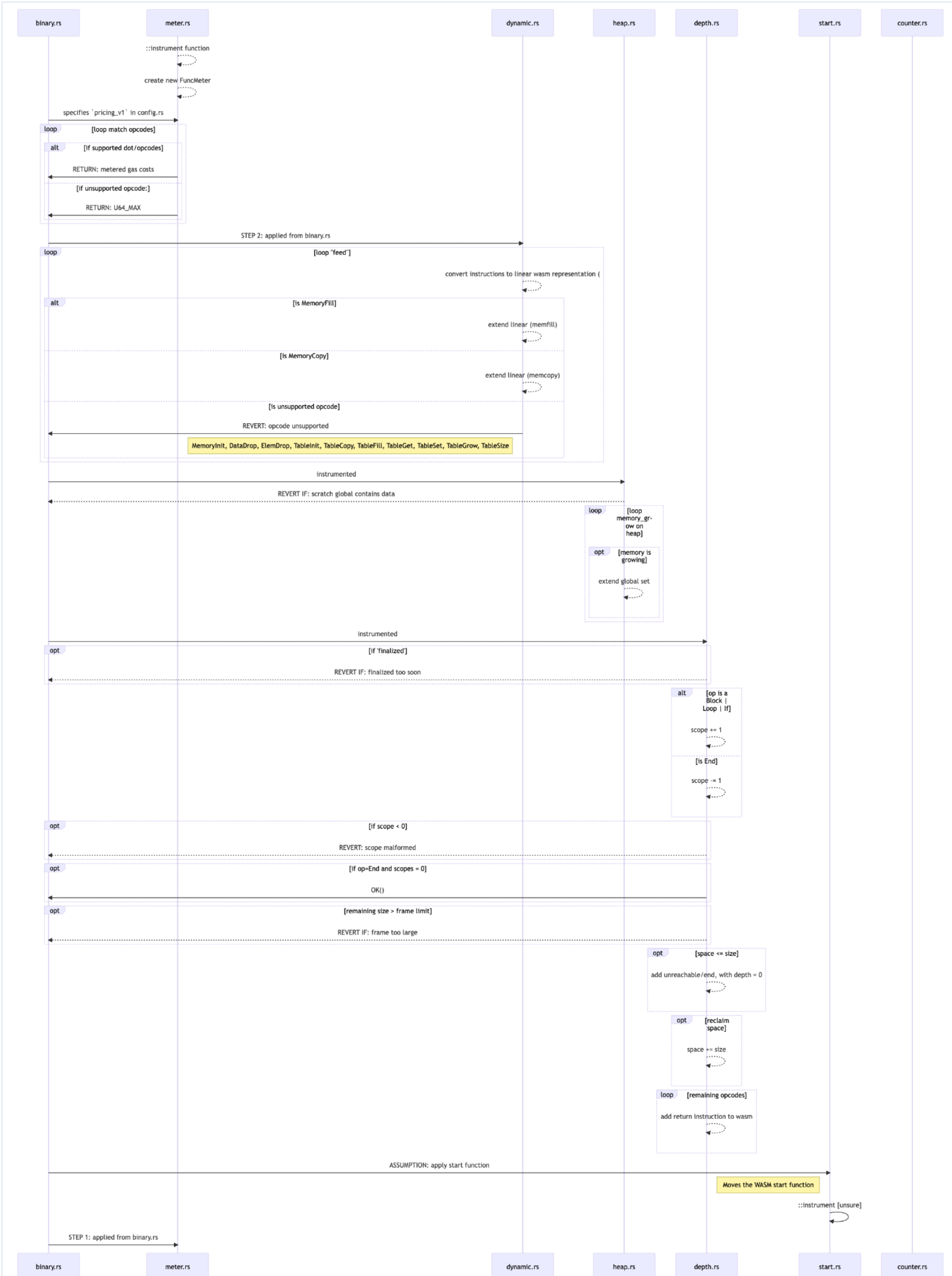


Figure C.2: A sequence diagram of the middleware used in the system

Figure C.3 shows the mermaid-js markdown used to render the flowcharts.

````
# WASM Instrumentation

```mermaid
sequenceDiagram;

participant caller


participant config.rs
caller ->> config.rs: compile version, price parameters, memory bounds, and debug
parameters
config.rs -->> config.rs: CompileConfig::store: saves parameter to memory
config.rs ->> config.rs: push meter, dygas, depth, bound, start middleware
note right of config.rs: see config.rs#L190-L194

participant binary.rs
caller ->> binary.rs: ::parseUser to instrument user wasm
binary.rs ->> binary.rs: ::instrument() function
loop Loop for all "codes"
    binary.rs -->> binary.rs: IF middleware, call ::instrument
    binary.rs -->> binary.rs: swap memory
    binary.rs -->> caller: if failed, wrap error
    binary.rs -->> binary.rs: apply [meter, dynamic meter, depth checker, heap
bound, start] in order
    binary.rs ->> caller: RETURN StylusData (ink_left, ink_status, depth_left,
footprint)
end
binary.rs -->> caller: REVERT IF: memory missing with export name
binary.rs -->> caller: REVERT IF: wasm does not fit into remaining memory
binary.rs -->> caller: REVERT IF: too many wasm instructions (antidos)
binary.rs -->> caller: REVERT IF: too many macro rules
binary.rs -->> caller: REVERT IF: "name" or "module name" is too long
binary.rs -->> caller: REVERT IF: is wasm start function
note left of binary.rs: can only have a single start function
binary.rs -->> caller: REVERT IF: missing stylus entrypoint
binary.rs -->> caller: REVERT IF: stylus entrypoint is not a function
binary.rs -->> caller: REVERT IF: entrypoint does not take ArbValueType::I32 input
and ArbValueType::I32 output
```

# Middlewares
``` mermaid
sequenceDiagram;

participant binary.rs

participant meter.rs
meter.rs -->> meter.rs: ::instrument function #L70
meter.rs -->> meter.rs: create new FuncMeter
binary.rs ->> meter.rs: specifies `pricing_v1` in config.rs
loop loop match opcodes
````
````

```
    alt if supported dot/opcodes
        meter.rs ->> binary.rs: RETURN: metered gas costs
    else if unsupported opcode:
        meter.rs ->> binary.rs: RETURN: U64_MAX
    end
end

participant dynamic.rs
binary.rs ->> dynamic.rs: STEP 2: applied from binary.rs
loop loop "feed"
    dynamic.rs -->> dynamic.rs: convert instructions to linear wasm representation
(#L105-L134)
    alt is MemoryFill
        dynamic.rs -->> dynamic.rs: extend linear (memfill)
    else is MemoryCopy
        dynamic.rs -->> dynamic.rs: extend linear (memcopy)
    else is unsupported opcode
        dynamic.rs ->> binary.rs: REVERT: opcode unsupported
        note left of dynamic.rs: MemoryInit, DataDrop, ElemDrop, TableInit,
TableCopy, TableFill, TableGet, TableSet, TableGrow, TableSize
    end
end

participant heap.rs
binary.rs ->> heap.rs: instrumented
heap.rs -->> binary.rs: REVERT IF: scratch global contains data
loop loop memory_grow on heap
    opt memory is growing
        heap.rs -->> heap.rs: extend global set
    end
end


participant depth.rs
binary.rs ->> depth.rs: instrumented
opt if 'finalized'
    depth.rs -->> binary.rs: REVERT IF: finalized too soon
end
alt op is a Block | Loop | If
    depth.rs -->> depth.rs: scope += 1
else is End
    depth.rs -->> depth.rs: scope -= 1
end
opt if scope < 0
    depth.rs -->> binary.rs: REVERT: scope malformed
end
opt if op=End and scopes = 0
    depth.rs ->> binary.rs: OK()
end
opt remaining size > frame limit
    depth.rs -->> binary.rs: REVERT IF: frame too large
end
opt space <= size
```

````
    depth.rs -->> depth.rs: add unreachable/end, with depth = 0
end
opt reclaim space
    depth.rs -->> depth.rs: space += size
end
loop remaining opcodes
    depth.rs -->> depth.rs: add return instruction to wasm
end

binary.rs ->> start.rs: ASSUMPTION: apply start function

participant start.rs
note left of start.rs: Moves the WASM start function
start.rs ->> start.rs: ::instrument [unsure]
binary.rs ->> meter.rs: STEP 1: applied from binary.rs

participant counter.rs
```
````

Figure C.3: The mermaid-js representations of the sequence diagrams in the appendix (figures C.1–C.2)
````

D. Code Quality Findings

Rust Code

- Some areas of the code write values to WASM memory (e.g., `block_coinbase`), while other areas read values (e.g., `block_gas_limit`). The need to write values in those cases is unclear.

```
pub(crate) fn block_coinbase<E: EvmApi>(mut env: WasmEnvMut<E>, ptr: u32) ->
MaybeEscape {
    let mut env = WasmEnv::start(&mut env, PTR_INK)?;
    env.write_bytes20(ptr, env.evm_data.block_coinbase)?;
    trace!("block_coinbase", env, &[], env.evm_data.block_coinbase)
}
pub(crate) fn block_gas_limit<E: EvmApi>(mut env: WasmEnvMut<E>) -> Result<u64,
Escape> {
    let mut env = WasmEnv::start(&mut env, 0)?;
    let limit = env.evm_data.block_gas_limit;
    trace!("block_gas_limit", env, &[], be!(limit), limit)
}
```

Figure D.1: The `block_coinbase` and `block_gas_limit` functions in `arbitrator/wasm-libraries/user-host-trait/src/lib.rs#L665–L682`

- The `contractCallImpl` function's `evmGas` variable (figure D.2) is overwritten in the `contract_call` function (figure D.3), which can lead to confusion.

```
func contractCallImpl(api usize, contract bytes20, data *rustSlice, evmGas *u64,
value bytes32, len *u32) apiStatus {
	closures := getApi(api)
	ret_len, cost, err := closures.contractCall(contract.toAddress(), data.read(),
uint64(*evmGas), value.toBig())
	*evmGas = u64(cost) // evmGas becomes the call's cost
	*len = u32(ret_len)
	if err != nil {
		return apiFailure
	}
	return apiSuccess
}
```

Figure D.2: The `contractCallImpl` function sets the `evmGas` variable. (`arbos/programs/native.go`)

```
fn contract_call(
    &mut self,
    contract: Bytes20,
    calldata: &[u8],
    gas: u64,
    value: Bytes32,
```

```
) -> (u32, u64, UserOutcomeKind) {
    let mut call_gas = gas; // becomes the call's cost
```

Figure D.3: The `contract_call` function in `arbitrator/arbutil/src/evm/api.rs`

- Functions do not always have the same ordering of return arguments (e.g., `static_call` and `create1`), which is an error-prone coding practice.

```
fn static_call(
    &mut self,
    contract: Bytes20,
    calldata: &[u8],
    gas: u64,
) -> (u32, u64, UserOutcomeKind) {
    {...omitted for brevity...}
    (return_data_len, call_gas, api_status.into())
}

fn create1(
    &mut self,
    code: Vec<u8>,
    endowment: Bytes32,
    gas: u64,
) -> (Result<Bytes20>, u32, u64) {
    {...omitted for brevity...}
    (result, return_data_len, call_gas)
}
```

Figure D.4: The `static_call` and `create1` functions in `arbitrator/arbutil/src/evm/api.rs`

- Some operations that can panic are undocumented. Additionally, some operations use `unwrap` instead of `expect` (e.g., `universal_test`), so reasons for erroring are unclear.

```
pub fn universal_test(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let item_clone = item.clone();
    let mut iter = item_clone.into_iter();
    let _ = iter.next().unwrap(); // fn
    {...omitted for brevity...}
}
```

Figure D.5: The `universal_test` function in `arbitrator/tools/wasmer/lib/api/macro-wasmer-universal-test/src/lib.rsL7-L36`

- The `jump_into_func` function includes an error message with a typo: `offest` should be `offset`.

```
pub fn jump_into_func(&mut self, module: u32, func: u32, mut args: Vec<Value>) ->
Result<()> {
    let Some(source_module) = self.modules.get(module as usize) else {
        bail!("no module at offest {}", module.red())
    };
    {...omitted for brevity...}
}
```

Figure D.6: The `jump_into_func` function in `arbitrator/prover/src/machine.rs#L1730–L1740`

- The checks in the `Machine` and `NativeInstance` `run_main` functions are reversed (figure D.7). Additionally, `Machine::run_main` and `Machine::from_user_path` should be marked as test functions with `#[cfg(test)]`. Make sure this tag is used consistently.

```
impl RunProgram for Machine {
fn run_main(&mut self, args: &[u8], config: StylusConfig, ink: u64) ->
Result<UserOutcome> {
   {...omitted for brevity...}
        if self.ink_left() == MachineMeter::Exhausted {
            return UserOutcome::OutOfInk;
        }
        if self.stack_left() == 0 {
            return UserOutcome::OutOfStack;
        }
        {...omitted for brevity...}
}

impl<E: EvmApi> RunProgram for NativeInstance<E> {
fn run_main(&mut self, args: &[u8], config: StylusConfig, ink: u64) ->
Result<UserOutcome> {
    {...omitted for brevity...}
    let status = match main.call(store, args.len() as u32) {
        Ok(status) => status,
        Err(outcome) => {
            if self.stack_left() == 0 {
                return Ok(OutOfStack);
            }
            if self.ink_left() == MachineMeter::Exhausted {
                return Ok(OutOfInk);
            }
        {...omitted for brevity...}
}
```

Figure D.7: The `Machine` and `NativeInstance` `run_main` functions in `arbitrator/stylus/src/run.rs`

- WASM compilation relies on `CallImport`, which is undocumented and is now deprecated in favor of `WasmImport`.

- The comment for the `get_return_data` function is incorrect; the highlighted part should say `vm.RETURNDATACOPY`.

```
/// Returns the EVM return data.
/// Analogous to `vm.RETURNDATASIZE`.
fn get_return_data(&mut self, offset: u32, size: u32) -> Vec<u8>;
```

Figure D.8: Misleading comment about retrieving return data (`stylus/arbitrator/arbutil/src/evm/api.rs#113–115`)

- The comment on the custom opcodes in `wavm.rs` indicates that more documentation exists in a file called "Custom opcodes.md." This file is not included in the repository. Either include the file in the repository or change the comment to link to the proper resource.

```
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash, Serialize, Deserialize)]
pub enum Opcode {
    Unreachable,
    Nop,
    {...omitted for brevity...}
    // Custom opcodes not in WASM. Documented more in "Custom opcodes.md".
    /// Custom opcode not in wasm.
    InitFrame,
```

Figure D.9: `stylus/arbitrator/prover/src/wavm.rs#L131`

- Various timeout durations are hard-coded across the codebase. Change them to constant values and document how the duration values were chosen.

```
impl RequestHandler<VecReader> for CothreadRequestor {
    fn handle_request(/* (...) */) -> (Vec<u8>, VecReader, u64) {
        // ...
        match self.rx.recv_timeout(Duration::from_secs(5)) { /* ... */ }


impl CothreadHandler {
    pub fn wait_next_message(&mut self) -> MaybeEscape {
        let msg = self.rx.recv_timeout(Duration::from_secs(10));
        // ...
    }

child_timeout: Duration::from_secs(15),
```

Figure D.10: Hard-coded timeouts in `stylus_backend.rs` and `machine.rs`

- The `call_user_func` function is never used outside of tests, as shown in the screenshot included in figure D.8, so it should be included only in test builds.

```
pub fn call_user_func(&mut self, func: &str, args: Vec<Value>, ink: u64) ->
```

```
Result<Vec<Value>> {
    self.set_ink(ink);
    self.call_function("user", func, args)
}
```

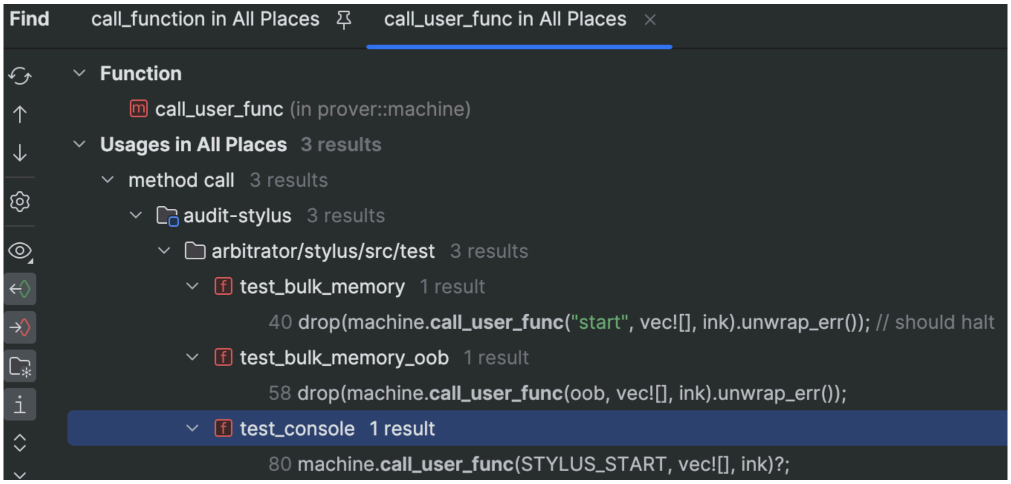


Figure D.11: The `call_user_func` function in `stylus/arbitrator/prover/src/machine.rs#L1796–1799`; a screenshot showing that the function is used only in tests

- Programs are currently not forced to be reactivated if a change in the Stylus version is detected during program execution; this could be used as a way to bypass the important system invariant that contracts being executed should have been activated under the current version. For instance, if the Stylus version is upgraded from a smart contract (e.g., a multisignature wallet or a DAO), then when the call to `ArbOwner` to set a new Stylus version returns, the contract under execution was never activated under the current version.

- The RETURN_DATA_COPY host I/O operation does not error out when reading out of bounds. The EVM equivalent will revert if `offsite + size` overflows or if `offsite + size` is greater than `len(return_data)`; this is a divergence that can lead to user mistakes.

```
fn read_return_data(
    &mut self,
    dest: GuestPtr,
    offset: u32,
    size: u32,
) -> Result<u32, Self::Err> {
    self.buy_ink(HOSTIO_INK + EVM_API_INK)?;

    // pay for only as many bytes as could possibly be written
    let max = self.evm_return_data_len().saturating_sub(offset);
    self.pay_for_write(size.min(max))?;
```

```
    let ret_data = self.evm_api().get_return_data();
    let ret_data = ret_data.slice();
    let out_slice = arbutil::slice_with_runoff(&ret_data, offset,
offset.saturating_add(size));

    let out_len = out_slice.len() as u32;
    if out_len > 0 {
        self.write_slice(dest, out_slice)?;
    }
    trace!(
        "read_return_data",
        self,
        [be!(offset), be!(size)],
        out_slice.to_vec(),
        out_len
    )
}
```

Figure D.12: `stylus/arbitrator/prover/src/machine.rs`

- The error cases shown in figure D.9 are not accompanied by clear error messages.

```
let mut asm_estimate: u64 = 512000;
{...omitted for brevity...}

let mut cached_init: u64 = 0;
{...omitted for brevity...}

let mut init = cached_init;
{...omitted for brevity...}

Ok(StylusData {
    ink_left: ink_left.as_u32(),
    ink_status: ink_status.as_u32(),
    depth_left: depth_left.as_u32(),
    init_cost: init.try_into()?,
    cached_init_cost: cached_init.try_into()?,
    asm_estimate: asm_estimate.try_into()?,
    footprint,
    user_main,
})
```

Figure D.13: Error cases in which `u64` *variables are cast into* `u16` *types, without proper error messages (*`stylus/arbitrator/prover/src/binary.rs#606–635`*)*

Solidity Code

- All the prefixes used during verification of Merkel trees in the OSP are included as the same string. Because they are all the same, consider using a global variable to hold the string prefix.

```
function proveLastLeaf(
```

```
    Machine memory mach,
    uint256 offset,
    bytes calldata proof
)
    internal
    pure
    returns (
        uint256 leaf,
        MerkleProof memory leafProof,
        MerkleProof memory zeroProof
    )
{
    string memory prefix = "Module merkle tree:";
    {...omitted for brevity...}
}

function executeLinkModule(
    ExecutionContext calldata,
    Machine memory mach,
    Module memory mod,
    Instruction calldata,
    bytes calldata proof
) internal pure {
    string memory prefix = "Module merkle tree:";
    {...omitted for brevity...}
}

function executeUnlinkModule(
    ExecutionContext calldata,
    Machine memory mach,
    Module memory,
    Instruction calldata,
    bytes calldata proof
) internal pure {
    string memory prefix = "Module merkle tree:";
    {...omitted for brevity...}
}
```

Figure D.14: Hard-coded strings in `stylus-contracts/src/osp/OneStepProverHostIo.sol#L392–483`

- The `_packBid` function truncates the upper 64 bits of the `uint256` bid value.

```
/// Creates a packed bid item
function _packBid(uint256 bid, uint64 index) internal pure returns (uint256) {
    return (bid << 64) | uint256(index);
}
```

Figure D.15: The bid value is truncated by shifting its bits. (`stylus-contracts/src/chain/CacheManager.sol#184–187`)

- Users could bid for space with `msg.value == 0`. Consider adding a minimum bid to prevent this.

Go Code

- There is a print statement in the production code, shown in figure D.11:

```
// Inserts a new item, returning true if already present.
func (p RecentWasms) Insert(item common.Hash, retain uint16) bool {
	if p.cache == nil {
		cache := lru.NewBasicLRU[common.Hash, struct{}](int(retain))
		p.cache = &cache
	}
	if _, hit := p.cache.Get(item); hit {
		println("hit!")
		return hit
	}
	p.cache.Add(item, struct{}{})
	return false
}
```

Figure D.16: A print statement is triggered when the cache is hit.
(`stylus-geth/core/state/statedb_arbitrum.go#275–287`)

E. Patch That Extends Tests and Assertions

This appendix contains code for the patch that will address TOB-STYLUS-19.

```
diff --git a/arbitrator/prover/src/binary.rs b/arbitrator/prover/src/binary.rs
index e4f7c2fc..aa8196bf 100644
--- a/arbitrator/prover/src/binary.rs
+++ b/arbitrator/prover/src/binary.rs
@@ -216,7 +216,7 @@ pub fn op_as_const(op: Operator) -> Result<Value> {
     }
 }

-#[derive(Clone, Debug, Default)]
+#[derive(Clone, Debug, Default, PartialEq)]
 pub struct FuncImport<'a> {
     pub offset: u32,
     pub module: &'a str,
@@ -306,7 +306,7 @@ pub fn parse<'a>(input: &'a [u8], path: &'_ Path) -> Result<WasmBinary<'a>> {
     };
     Validator::new_with_features(features)
         .validate_all(input)
-        .wrap_err_with(|| eyre!("failed to validate {}", path.to_string_lossy().red()))?;
+        .map_err(|err| eyre!("failed to validate {}: {}", path.to_string_lossy().red(), err))?;

     let mut binary = WasmBinary::default();
     let sections: Vec<_> = Parser::new(0).parse_all(input).collect::<Result<_, _>>()?;
diff --git a/arbitrator/prover/src/test.rs b/arbitrator/prover/src/test.rs
index 44a8dff0..c3bda441 100644
--- a/arbitrator/prover/src/test.rs
+++ b/arbitrator/prover/src/test.rs
@@ -3,8 +3,15 @@

 #![cfg(test)]

+use fnv::FnvHashMap as HashMap;
 use crate::binary;
 use std::path::Path;
+use wasmparser::MemoryType;
+use crate::binary::ExportKind::{Func, Global, Memory};
+use crate::programs::prelude::CompileConfig;
+use crate::binary::{ExportMap, NameCustomSection, WasmBinary};
+use crate::value::ArbValueType::I32;
+use crate::value::{FunctionType, Value};

 fn as_wasm(wat: &str) -> Vec<u8> {
     let wasm = wasmer::wat2wasm(wat.as_bytes());
@@ -20,7 +27,11 @@ pub fn reject_reexports() {
             (func $should_reject (export "some_hostio_func") (param) (result))
         )"#,
     );
-    let _ = binary::parse(&wasm, Path::new("")).unwrap_err();
+    let expected_error_msg = "binary exports an import with the same name \u{1b}[31;1msome_hostio_func\u{1b}[0;0m";
+    assert_eq!(
+        binary::parse(&wasm, Path::new("")).unwrap_err().to_string(),
+        expected_error_msg
+    );

     let wasm = as_wasm(
         r#"
@@ -29,7 +40,10 @@ pub fn reject_reexports() {
             (global $should_reject (export "some_hostio_func") f32 (f32.const 0))
         )"#,
     );
-    let _ = binary::parse(&wasm, Path::new("")).unwrap_err();
+    assert_eq!(
+        binary::parse(&wasm, Path::new("")).unwrap_err().to_string(),
+        expected_error_msg
+    );
 }

 #[test]
@@ -41,7 +55,17 @@ pub fn reject_ambiguous_imports() {
             (import "vm_hooks" "some_import" (func (param i64) (result i64 i32)))
         )"#,
     );
```

```
-     let _ = binary::parse(&wasm, Path::new("")).unwrap();
+     let bin = binary::parse(&wasm, Path::new("")).unwrap();
+
+     // Note: import names are not necessarily unique
+     // https://webassembly.github.io/spec/core/syntax/modules.html#imports
+     assert_eq!(bin.imports.len(), 2);
+     for i in 0..2 {
+         let imp = &bin.imports[i];
+         assert_eq!(imp.offset, 0);
+         assert_eq!(imp.module, "vm_hooks");
+         assert_eq!(imp.name, "some_import");
+     }

      let wasm = as_wasm(
          r#"
@@ -50,5 +74,134 @@ pub fn reject_ambiguous_imports() {
              (import "vm_hooks" "some_import" (func (param i32) (result)))
          )"#,
      );
-     let _ = binary::parse(&wasm, Path::new("")).unwrap_err();
+     assert_eq!(
+         binary::parse(&wasm, Path::new("")).unwrap_err().to_string(),
+         "inconsistent imports for \u{1b}[31;1mvm_hooks\u{1b}[0;0m \u{1b}[31;1m\"some_import\"\u{1b}[0;0m"
+     );
+}
+
+
+#[test]
+pub fn parse_user_expect_missing_export_with_name_user_entrypoint() {
+     let wasm = as_wasm(
+         r#"
+     (module
+     (memory 0 0)
+     (export "memory" (memory 0)))
+"#,
+     );
+     let compile = CompileConfig::version(0, true);
+     let result = WasmBinary::parse_user(&wasm, 1, &compile);
+     assert_eq!(
+         result.err().expect("error expected").to_string(),
+         "missing export with name \u{1b}[31;1muser_entrypoint\u{1b}[0;0m"
+     );
+}
+
+#[test]
+pub fn parse_user_export_name_within_max_length() {
+     let longest_acceptable_export_name = "A".repeat(500);
+     let wasm = as_wasm(
+         &format!(r#"
+     (module
+     (memory 0 0)
+     (export "memory" (memory 0))
+     (type $void (func (param) (result)))
+     (func (export "user_entrypoint") (param $args_len i32) (result i32)
+         i32.const 0
+     )
+     (func (export "{}") (type $void))
+     )
+"#, longest_acceptable_export_name),
+     );
+     let compile = CompileConfig::version(0, true);
+     let result = WasmBinary::parse_user(&wasm, 1, &compile);
+
+     assert!(result.is_ok());
+     let (binary, _data, pages) = result.unwrap();
+     // TODO/FIXME: Ideally, this should be:
+     // assert_eq!(binary, WasmBinary{ ... });
+     // but it requires e.g. 'Element' struct to be Eq/PartialEq
+     assert_eq!(binary.types, vec![FunctionType { inputs: vec![], outputs: vec![] }, FunctionType { inputs: vec![I32], 
outputs: vec![I32] }]);
+     assert_eq!(binary.imports, vec![]);
+     assert_eq!(binary.tables, vec![]);
+     assert_eq!(binary.memories, vec![MemoryType { memory64: false, shared: false, initial: 0, maximum: Some(0) }]);
+     assert_eq!(binary.globals, vec![Value::I64(0), Value::I32(0), Value::I32(0), Value::I32(0)]);
+
+     let mut expected_exports : ExportMap = Default::default();
+     expected_exports.insert("stylus_ink_left".to_string(), (0u32, Global));
+     expected_exports.insert(longest_acceptable_export_name.clone(), (1, Func));
+     expected_exports.insert("memory".into(), (0, Memory));
+     expected_exports.insert("stylus_ink_status".into(), (1, Global));
```

```
+    expected_exports.insert("stylus_scratch_global".into(), (2, Global));
+    expected_exports.insert("user_entrypoint".into(), (0, Func));
+    expected_exports.insert("stylus_stack_left".into(), (3, Global));
+
+    assert_eq!(binary.exports, expected_exports);
+    assert_eq!(binary.start, None);
+    assert_eq!(binary.elements.len(), 0);
+
+    assert_eq!(binary.codes.len(), 2);
+
+    let mut expected_functions = HashMap::default();
+    expected_functions.insert(1u32, longest_acceptable_export_name);
+    expected_functions.insert(0u32, "user_entrypoint".to_string());
+
+    assert_eq!(binary.names, NameCustomSection {
+        module: "user".to_string(),
+        functions: expected_functions
+    });
+    assert_eq!(pages, 0);
+    // TODO/FIXME: Assert binary.codes and _data
+}
+
+
+#[test]
+pub fn parse_user_export_name_too_long() {
+    let longest_acceptable_export_name = "A".repeat(501);
+    let wasm = as_wasm(
+        &format!(r#"
+    (module
+    (memory 0 0)
+    (export "memory" (memory 0))
+    (func (export "{}") (void) (void))
+    )
+"#, longest_acceptable_export_name),
+    );
+    let compile = CompileConfig::version(0, true);
+    let result = WasmBinary::parse_user(&wasm, 1, &compile);
+    assert_eq!(
+        result.err().expect("error expected").to_string(),
+        "wasm \u{1b}[31;1mname\u{1b}[0;0m too long: \u{1b}[31;1m501\u{1b}[0;0m > \u{1b}[31;1m500\u{1b}[0;0m"
+    );
+}
 }
```

Figure E.1: Patch for tests

F. Toward an Automated Fuzzing Process

For this engagement, we created a fuzz test that uses bolero, an in-process, coverage-guided, evolutionary fuzzing engine that works with Rust code. This test covers the parsing and processing of WASM programs, which take untrusted inputs. We integrated the fuzz test into the tests of the `arbitrator/stylus` crate (i.e., added a new test to `tests/native.rs`). Figure F.1 shows the first part of the `fuzz_wasm` test:

```
#[test]
fn fuzz_wasm() {
    bolero::check!().for_each(|data: &[u8]| {

      let exports_test_case = r#"
        (module
                (func (export "user_entrypoint") (param i32) (result i32)
                unreachable
                )
        )
        "#;
      let available_imports = r#"
        (module
          (import "vm_hooks" "pay_for_memory_grow" (func $pay_for_memory_grow (param
i32)))
          (import "vm_hooks" "read_args"    (func $read_args    (param i32)))
          (import "vm_hooks" "write_result" (func $write_result (param i32 i32)))
          (import "vm_hooks" "exit_early"   (func $exit         (param i32)))
          (import "vm_hooks" "msg_value"           (func $msg_value     (param
i32)))
          (import "vm_hooks" "call_contract"       (func $call_contract (param i32
i32 i32 i32 i64 i32) (result i32)))
          (import "vm_hooks" "block_coinbase" (func $block_coinbase         (param
i32)))
          (import "vm_hooks" "contract_address" (func $contract_address
(param i32)))
          (import "vm_hooks" "chainid"      (func $chainid         (result i64)))
          (import "vm_hooks" "evm_gas_left" (func $evm_gas_left    (result i64)))
        )
      "#;

      let wasm_exports = wat::parse_str(exports_test_case).unwrap();
      let wasm_imports = wat::parse_str(available_imports).unwrap();

      let mut unstructured = Unstructured::new(data);
      let mut config = Config::arbitrary(&mut unstructured).expect("arbitrary
config");
      config.min_funcs = 1;
      config.max_funcs = 5;
      config.max_memories = 1;
      config.min_memories = 1;
      config.bulk_memory_enabled = false;
```

```
      config.multi_value_enabled = false;
      config.exports = Some(wasm_exports.clone());
      config.available_imports = Some(wasm_imports.clone());
      …
```

Figure F.1: The header of the fuzz test

In order to run the test efficiently, Stylus needs well-formed WASM binaries. The use of raw bytes to feed the fuzzer is absolutely impractical because WASM modules have rather strict constraints. Therefore, we decided to use `wasm-smith` to craft valid WASM binaries. While we could have used the vanilla version of the tool, the results would not have been good because the WASM binaries need certain imports and exports that vanilla `wasm-smith` would have been extremely unlikely to include by chance. Therefore, we had to fork `wasm-smith` to add a few small features.

Once a module is properly created, native execution is used to quickly run the binary:

```
      if let Ok(module) = Module::new(config, &mut unstructured)  {
      let wasm_bytes = module.to_bytes();
      let mut file_native = NamedTempFile::new().unwrap();
      file_native.write_all(&wasm_bytes).unwrap();
      file_native.flush().unwrap();

      let mut compile = my_test_compile_config(false);
      let config = my_uniform_cost_config();
      if let Ok(mut native) =
NativeInstance::new_linked(file_native.path().display().to_string(), &compile,
config) {
          let ink = 1_000_000;
          native.set_meter_data();
          native.set_ink(ink);
          let args = vec![1];
          run_native(&mut native, &args, ink);
```

Figure F.2: Native execution of the WASM binary

Finally, after native execution, the WASM binary can be run in prover mode:

```
let mut file_machine = NamedTempFile::new().unwrap();
file_machine.write_all(&wat.as_bytes()).unwrap();
file_machine.flush().unwrap();

let maybe_machine = Machine::from_user_path(file_machine.path(), &compile);
match maybe_machine {
   Ok(mut machine) => {
   run_machine(&mut machine, &args, config, ink);
   check_instrumentation(native, machine);
   }
   Err(err) => {
       println!("{}", err);
```

```
    }
}
```

Figure F.3: Prover execution of the binary

This part of the fuzzing test is very slow (with single executions usually taking around 20 seconds). The produced binaries to test are small, so we suspect that most of the processing time comes from the processing and conversion of libraries (e.g., soft float libraries). This code will also check that the native and instrumented machine consumes the same amount of ink and memory stacks.

With all these steps completed, the fuzzer can then be run using the following command:

`$ cargo bolero test test::native::fuzz_wasm -s NONE`

After the build finishes, `bolero` will immediately start running the fuzzing campaign. ASan can be enabled by removing `-s NONE` from the command, but this results in slower execution times and may trigger some crashes from Wasmer (e.g., TOB-STYLUS-8, TOB-STYLUS-9).

When this command is used, `bolero` will run for as long as needed until it triggers a failure. We recommend monitoring the values on screen to make sure the fuzzer explores as many code paths as possible.

Note that this fuzzer will not directly use the coverage provided by the execution of the WASM code compiled as native. Instead, it will use the coverage of the underlying Rust implementation. Coverage of the compiled WASM code will have to be implemented in order to inspect it and to increase the effectiveness of the fuzzing campaign.

Recommendations

The testing code needs to be refactored to be as close as possible to the production code. In that regard, we have a few recommended changes:

- The use of native execution during testing fails to validate WASM binaries as part of the instrumentation executed during activation of the binaries. This results in divergences in executions when the native execution contains an unsupported WASM feature (e.g., vectorized operations). Make sure the validation after activation is performed exactly the same for different modes.

- The test's prover execution code uses the `user-test` library instead of the production code. It should be changed to use the production code so that it more closely matches the implementation.

- The testing code also contains some testing code related to EVM handling (e.g., calling another contract). This code should be changed to use the equivalent from the production code. While it is not critical to have the same behavior, using as much of the production code as possible is useful for finding issues in Stylus.

Moving forward, we recommend that Offchain implement a “fuzzer-friendly mode” that avoids performing very CPU-intensive operations that can be skipped during a fuzzing campaign.

Additionally, consider adapting specific WASM tools such as `xsmith` in order to trigger more interesting and complex behavior that `wasm-smith` cannot easily reach, particularly for testing optimized code.

Integrating Fuzzing and Coverage Measurement into the Development Cycle

Once the fuzzing procedure has been tuned to be fast and efficient, it should be properly integrated into the development cycle to catch bugs. We recommend adopting the following procedure to integrate fuzzing using a CI system:

1. After the initial fuzzing campaign, save the corpora that is generated for every test.
2. For every internal development milestone, new feature, or public release, rerun the fuzzing campaign for at least 24 hours starting with the current corpora for each test.
3. Update the corpora with the new inputs generated.

Note that, over time, the corpora will come to represent thousands of CPU hours of refinement and will be very valuable for guiding efficient code coverage during fuzz testing. However, an attacker could also use them to quickly identify vulnerable code. To mitigate this risk, we recommend keeping the fuzzing corpora in an access-controlled storage location rather than a public repository. Some CI systems allow maintainers to keep a cache to accelerate building and testing. The corpora could be included in such a cache if they are not very large. For more on fuzz-driven development, see the CppCon 2017 talk given by Google’s Kostya Serebryany.

G. Recommendations for Improving Integration Tests

In this appendix, we provide recommendations for making improvements to the tests found in `system_tests/program_tests.go` and for including additional test cases. Note that additional assertions may need to be performed in some of the test cases to cover all edge cases.

- **The test "`Checking success (Rust => Solidity => Rust)`" should assert the following:**
    - A single transaction was executed.
    - Two internal transactions were executed.
    - The transaction returned the expected Keccak data.
- **Add a test to ensure that static call properties are preserved (Solidity => Stylus => Solidity):**
    - Have a Solidity contract perform a static call into a Stylus program.
    - Have the Stylus program call another Solidity contract that tries to modify the state.
    - Verify that a revert occurs during the Stylus program's call to the other Solidity contract.

  An additional version of this test could perform a write to storage within the Stylus program instead of calling into Solidity.

  Note that a test case already exists from the Stylus counterpart.
- **Add a test to ensure that the self-destruct opcode cannot be used to destroy Stylus programs (Stylus => Solidity).** Stylus programs cannot contain the self-destruct opcode; however, they use `delegatecall`, which could make self-destruction possible:
    - Have a Stylus program call `delegatecall` on a Solidity program to self-destruct the Stylus program.
    - Have the test ensure that the Stylus program is no longer callable and that it no longer has code.
    - Have the test verify that the Stylus program cannot be reactivated.

  Consider using EIP-6780 for this test.

Note that additional test cases could verify that the Stylus program that has been self-destructed is still callable within the same transaction.

- **Add a test for `delegatecall` functionality (Solidity => Stylus && Stylus => Solidity):**
    - Create a program with some values in its storage and have it call `delegatecall` on a program of a different kind (e.g., Stylus or Solidity) that reads those values from the storage and returns them.
    - Have the test check that the returned values match the expected ones.
- **Add a test to ensure that low-level Solidity call properties are preserved (Solidity => Stylus && Stylus => Solidity):**
    - Have a Solidity contract perform a low-level call to a Stylus program with no code. The test should ensure that the low-level call succeeds.
    - Have a Stylus program perform a low-level call to a Solidity program with no code or EOA. The test should ensure that the call succeeds.
- **Add a test to determine whether destroyed programs can be redeployed with `CREATE2` (Solidity || Stylus => Stylus => Solidity):**
    - Have a Solidity (or Stylus) contract deploy a new Stylus program through `CREATE2`.
    - Have the newly created Solidity (or Stylus) program call `delegatecall` on a Solidity contract to execute the self-destruct opcode on itself.
    - Have the test repeat the first step to verify that the self-destructed program can be redeployed.
    - Have the test ensure the contract is callable after the redeployment.

  Consider using EIP-6780 for this test.
- **Add tests to ensure reasons for reverts are preserved across environments (Stylus => Solidity && Solidity => Stylus):**
    - Have a Stylus program perform calls to a Solidity program, and have those calls revert for different reasons (e.g., panic due to overflow, panic due to out-of-bounds read) and in different ways (e.g., a string stating the reason, a custom error). Have the test ensure that those errors are correctly preserved by returning them and comparing them against their expected values. Create the same test case for a Solidity program calling a Stylus program.
- **Add various tests to check out-of-gas/ink cases such as the following:**

- From Rust directly
- From Rust calling to Rust
- From Solidity calling to Rust
- From Solidity calling to Rust calling to Rust again
- From Rust calling to Solidity calling to Rust
- From Rust calling to Solidity calling to Rust and calling to Rust again

- **Add a test of the stack frame limit:**
  - The test should verify that the stack depth for Solidity programs will never exceed 1,023 when called by a Stylus program. Starting from a Stylus program, have the program call a Solidity contract, which would then call the Stylus program, and so on.
  - The test should fail if the stack depth exceeds 1,023 Solidity frames.