Security Research Labs

Frontier Baseline Security Assurance

Threat model and hacking assessment report

v1.0, 24 May 2025

Prepared for:
Polkadot Assurance Legion

27 pages

Content

Disclaimer

This report describes the findings and core conclusions derived from the audit carried out by Security Research Labs within the timeframe and scope detailed in Chapter 3.

Please note that this report does not guarantee that all existing security vulnerabilities were discovered in the codebase exhaustively. Following all suggestions may not ensure future code to be bug free.

Version:	v1.0
Client:	Polkadot Assurance Legion
Date:	24 May 2025
Audit Team:	Daniel Schmidt Aarnav Bos Cayo Fletcher-Smith Kevin Valerio

Timeline

Security Research Labs performed the Frontier source code security assessment. The analysis completed within 8 weeks, starting from 4th of March 2025.

Date	Event
4 March 2025	Project Kickoff
30 April 2025	Report for the baseline security check delivered
24 May 2025	Final Report delivered (this document)

Table 1: Audit timeline

Integrity Notice

1 Executive summary

1.1 Engagement overview

This report documents the results of a baseline security assurance audit of Frontier that Security Research Labs performed from March to April 2025. Security Research Labs consultants have been providing specialized audit services for Polkadot and Polkadot SDK-based projects since 2019. During this study, Frontier provided access to relevant documentation and effectively supported the research team. We verified the protocol design and relevant available source code of Frontier.

This audit focused on assessing Frontier's codebase for resilience against hacking and abuse scenarios. Key areas of scrutiny included PoV estimation, gasometer, EVM, RPC, runtime configuration, account mapping and precompiles. The testing approach combined static and dynamic analysis techniques, leveraging both automated tools and manual inspection.

We prioritized reviewing critical functionalities and conducting thorough security tests to ensure the robustness of Frontier's platform. We collaborated closely with Frontier, utilizing full access to source code and documentation to perform a rigorous assessment.

1.2 Observations and risk

The research team identified several issues ranging from high to informational level severity. No critical issues were identified during the audit. Frontier partially acknowledged these issues and is actively working on remediation in cooperation with us.

1.3 Recommendations

In addition to mitigating the remaining open issues, Security Research Labs recommends applying additional resources to developing meaningful documentation both throughout the codebase and via external sources. Specific focus should be given to configuration options and complex processes that may otherwise be ambiguous to developers. Furthermore, alongside improving the test coverage, we encourage the continued maintenance of a specialized Frontier fuzzing campaign to help identify hard to reach edge cases.

2 Evolution suggestions

To ensure that Frontier is secure against further unknown or yet undiscovered threats, we recommend considering the following evolution suggestions and best practices described in this section.

2.1 Secure development improvement suggestions

We recommend further strengthening the security of the blockchain by implementing the following recommendations:

Perform threat modeling. Threat modeling for all new features and major updates before coding promotes better code security. This practice lets developers identify potential security threats and vulnerabilities early in the design phase, enabling them to implement appropriate mitigations from the outset. Including the threat model in the pull request description ensures that the entire team is aware of the identified risks and the measures taken to address them, promoting a proactive security culture and enhancing the overall robustness of the codebase. Additionally, it helps the audit team to identify gaps in the threat model and focus their assessment.

Use static analysis. Static analysis tools help detect security flaws in the codebase, thus improving code security. These tools, such as Dylint and Semgrep for the Rust ecosystem, analyze code without executing it. They identify vulnerabilities, coding errors, and compliance issues early in the development process. This proactive approach helps developers address potential security issues before they reach production, ensuring a more secure and reliable codebase.

Perform dynamic analysis. Continuously improving and developing harnesses for critical components is essential to identify security vulnerabilities and business logic issues. By employing invariants, fuzzing tests can effectively uncover subtle flaws that might otherwise go unnoticed. This is especially critical in a hardened codebase like Frontier, where bugs often emerge from the complex interplay of various modules, resulting in logical flaws that traditional testing overlooks.

The Polkadot codebase exemplifies this approach by utilizing multiple fuzzing harnesses based on the *substrate-runtime-fuzzer*. This demonstrates how comprehensive and targeted fuzz testing can significantly enhance the security and reliability of complex systems. For details, see the *substrate-runtime-fuzzer* [1].

Memory and computation usage analysis. Alongside continuous dynamic analysis through fuzzing, we recommend periodically performing execution analyses on the fuzzing corpus seeds to identify spikes in computation and catch undervalued processes. This practice may be implemented to ensure the robustness of existing benchmarks.

Account for edge cases in tests. Most test cases are reliant on artificial genesis data generated to test the desired module. This genesis data is generally forcefully loaded irreflective of the execution flow of on-chain operations. This approach fails to catch bugs in the execution flow necessary to initialize the genesis data itself. An example of this are the tests surrounding smart contract accounts, where fixed bytecode is directly inserted within storage instead of being deployed via a creation transaction. This fails to catch bugs and edge-cases within the creation itself, as illustrated in finding S3-55: Constructing smart contract can bypass precompile address bounding. We recommend improving tests to simulate the true flow of on-chain operations where possible.

Use safe operation alternatives. Adhering to safe default operations is strongly recommended. This includes replacing unsafe mathematical operations with safer alternatives, avoiding silent truncation during data conversions, and favoring error propagation over unwinding. In most cases, adopting

these safer practices doesn't have any significant downside. As demonstrated in Section 5, many of the identified issues could have been prevented by following the established best practices.

2.2 Address currently open security issues

We recommend addressing already-known security issues with timely updates. Even if an open issue has a limited impact, an attacker might use it as part of their exploitation chain, which may have a more severe impact on third-party chains relying on Frontier's security. This is especially true considering this codebase is currently live in many high-profile projects.

2.3 Further recommended best practices

Integrate Ethereum End-to-End Tests. Ethereum execution clients maintain a comprehensive suite of end-to-end test cases that can be continuously leveraged by Frontier. While Frontier does not implement a one-to-one feature set with Ethereum, integrating these tests would likely surface bugs early in the development lifecycle, improving overall software robustness.

Documentation. Documentation for Frontier and the EVM code was missing or incomplete. This made it challenging to understand the code without consulting the development team. Accurate documentation ensures that developers, auditors, and other stakeholders can comprehend the codebase without necessarily consulting the development team. To achieve this, establish a practice of updating the documentation concurrently with any code changes. Incorporating documentation verification into the code review process can help detect discrepancies early.

Regular updates. New released dependencies, as shown in Section 5. may contain fixes for critical security issues. Since Frontier is a product that heavily relies on various dependencies, updates to the latest version should be integrated as soon as possible. Many crates do not mark security fixes as such and hence updating crates used for important functionality early is highly recommended.

3 Motivation and scope

This report presents the results of the security audit for Frontier from March to April 2025.

Frontier aims to provide an Ethereum compatibility layer for the Polkadot ecosystem by implementing a broad Ethereum feature set. This vision has been fundamentally achieved with the following key features:

1. Support for Ethereum-style RPC calls to allow existing Ethereum applications to be compatible with Polkadot SDK based platforms.
2. Mapping for existing Substrate accounts to the 20-byte Ethereum address format which allows users and smart contract applications to interact with accounts uniformly between both Ethereum and Polkadot SDK platforms.
3. Integration of runtime gas metering to emulate the transaction fee mechanisms present in the Ethereum blockchain, while remaining compliant with the Substrate weight system. This allows Solidity smart contracts to exist on Frontier integrated Polkadot SDK based blockchains, without prior need for benchmarking.
4. Implementation of an extensive precompile feature set, mapping core Polkadot SDK pallets to Solidity interfaces accessible via Ethereum style calls.

Security Research Labs collaborated with the Frontier development team to create an overview containing the modules in scope and their audit priority. The in-scope components and their assigned priorities are reflected in Table 2. During the audit, Security Research Labs used threat modelling to guide our efforts on exploring potential security flaws and realistic attack scenarios.

Repository	Priority	Component(s)
Frontier [2]	High	fp-evm fp-storage pallet-evm pallet-ethereum
	Medium	pallet-dynamic-fee fp-consensus fc-consensus fc-db fc-mapping-sync fc-rpc-core fp-rpc fc-rpc
EVM [3]	High	evm-core evm-runtime evm-gasometer

Table 2: In-scope Frontier`s components with audit priority

4 Methodology

We applied the following four-step methodology when performing feature for Frontier: (1) threat modeling, (2) security design coverage checks, (3) implementation baseline check, and finally (4) remediation support.

4.1 Threat modeling and attacks

The goal of the threat model framework is to determine specific areas of risk in Frontier. Familiarity with these risk areas can provide guidance for the design of the implementation stack, the actual implementation of the stack, as well as security testing. This section introduces how risk is defined and provides an overview of the identified threat scenarios.

The risk level is categorized into low, medium, and high and considers both the hacking value and the damage that could result from successful exploitation. The risk of a threat scenario is calculated by the following formula:

$$Risk = Damage \times Hacking\ Value = \frac{Damage \times Incentive}{Effort}$$

The *Hacking Value* is similarly categorized into low, medium, and high and considers the incentive of an attacker, as well as the effort required by an adversary to successfully execute the attack. The hacking value is calculated as follows:

$$Hacking\ Value = \frac{Incentive}{Effort}$$

While incentive describes what an adversary might gain from performing an attack successfully, effort estimates the complexity of this same attack. The degrees of incentive and effort are defined as follows:

Incentive:

- Low: Attacks offer the hacker little to no gain from executing the threat
- Medium: Attacks offer the hacker considerable gains from executing the threat
- High: Attacks offer the hacker high gains by executing this threat

Easiness:

- High: Attacks are easy to execute. They require neither elaborate technical knowledge nor considerable amounts of resources
- Medium: Attacks are difficult to execute. They might require bypassing countermeasures, the use of expensive resources, or a considerable amount of technical knowledge
- Low: Attacks are difficult to execute. The attacks might require in-depth technical knowledge, vast amounts of expensive resources, bypassing countermeasures, or any combination of these factors

Incentive and Easiness are divided according to Table 3.

Hacking Value/Likelihood	Low Incentive	Medium Incentive	High Incentive
Low Easiness	Low	Medium	Medium
Medium Easiness	Medium	Medium	High
High Easiness	Medium	High	High

Table 3: Hacking value measurement scale

Hacking scenarios are classified by the risk they pose to the system. Conversely, the *Damage* describes the negative impact that a given attack, if performed successfully, would have on the victim. The degrees of damage are defined as follows:

Damage:

- Low: Risk scenarios would cause negligible damage to the Frontier network
- Medium: Risk scenarios pose a considerable threat to Frontier‘s functionality as a network
- High: Risk scenarios pose an existential threat to Frontier network functionality

Damage and Hacking Value are divided according to Table 4.

Risk	Low hacking value	Medium hacking value	High hacking value
Low damage	Low	Medium	Medium
Medium damage	Medium	Medium	High
High damage	Medium	High	High

Table 4: Risk measurement scale

After applying the framework to the Frontier system, different threat scenarios according to the CIA triad were identified.

The CIA triad describes three security promises that can be violated by a hacking attack, namely confidentiality, integrity, and availability.

Confidentiality:

Confidentiality threat scenarios concern sensitive information regarding the blockchain network and its users. Confidentiality threat scenarios include, for example, attackers abusing information leaks to steal native tokens from nodes participating in a Frontier reliant project and claiming the assets for themselves.

Integrity:

Integrity threat scenarios aim to disrupt the functionality of a Frontier dependent project by undermining or bypassing the rules that ensure that Frontier operations are fair and equal for each participant. Undermining Frontier‘s low-level integrity often comes with a high monetary incentive. For example, an attacker may abuse double-spend bugs or precompile logic to enact unique monetary exploits. Other threat scenarios that do not yield an immediate monetary reward could rather damage Frontier‘s functionality and, in turn, its reputation. For example, manipulating fee calculation or abusing low-level inconsistencies within Frontier’s Ethereum-virtual-machine.

Availability:

Availability threat scenarios refer to compromising the availability of data stored by the Frontier network as well as the availability of the network itself to process normal transactions. Important threat scenarios regarding the availability for blockchain systems include denial-of-service (DoS), stalling the transaction queue, and storage bloating.

Table 5 provides a high-level overview of the hacking risks concerning Frontier with the identified example threat scenarios and attacks, as well as their respective hacking value and effort. The complete list of threat scenarios identified along with attacks that enable them is provided in the threat model deliverable. This list can serve as a starting point for the Frontier developers to guide their security outlook for future feature implementations. By thinking in terms of threat scenarios and attacks during code review or feature ideation, many issues can be caught or even avoided altogether.

The threats were classified using the CIA security triad model, mapping threats to the areas: (1) Confidentiality, (2) Integrity, and (3) Availability.

Security promise	Hacking value	Example threat scenarios	Hacking effort	Example attack ideas
Confidentiality	Medium	An attacker is able to compromise a user's private key	High	A node's private key is leaked through an insecure RPC call
Integrity	High	A malicious node is able to manipulate the storage of a specific smart contract	Medium	SSTORE is able to write into the context of other smart contracts due to a collision, potentially leading to the manipulation of a smart contract state that is not owned by the user, resulting in adverse side effects
Availability	High	An attacker is able to execute transactions on the chain without paying adequate fees	Low	Certain opcodes in the EVM are undervalued, allowing an attacker to execute these opcodes many times at minimal cost but with significant resource consumption

Table 5: Risk overview

4.2 Security design coverage check.

Next, we reviewed the Frontier design for coverage against relevant hacking scenarios. For each scenario, we have investigated the following two aspects:

a. **Coverage**. Is each potential security vulnerability sufficiently covered by our audit?

b. **Underlying assumptions**. Which assumptions must hold true for the design to effectively reach the desired security goal?

4.3 Implementation check

As a third step, we tested the current Frontier implementation for openings whereby any of the defined hacking scenarios could be executed.

To effectively review the Frontier codebase, we derived our code review strategy based on the threat model that we established as the first step. For each identified threat, hypothetical attacks were developed and mapped to their corresponding threat category, as outlined in Chapter 4.1.

Prioritizing risk, the code was assessed for present protection against the respective threats and attacks as well as the vulnerabilities that make these attacks possible. For each threat, we:

1. Identified the relevant parts of the codebase, for example, relevant pallets and the Ethereum virtual machine

2. Identified viable strategies for the code review. We performed manual code audits, fuzz testing, and manual where appropriate

3. Ensured the code did not contain any vulnerabilities that could be used to execute the respective attacks. Otherwise, we ensured that sufficient protection measures against specific attacks were present

4. Immediately reported any vulnerability that was discovered to the development team along with suggestions around mitigations

We carried out a hybrid strategy combining code review, static tests, and dynamic tests (e.g., fuzz testing) to assess the security of the Frontier codebase.

While static and dynamic testing establishes a baseline assurance, the focus of this audit was on manual code review of the Frontier codebase to identify logic bugs, design flaws, and best practice deviations. We reviewed the Frontier and EVM repositories which contain changes up to commit *4dddde8 from the 20^{s} of February 2025* for Frontier and the EVM up to commit *6d86fe2* from *the 18^{s} of January 2025*. We aimed to trace the intended functionality of the modules in scope and to assess whether an attacker can bypass/misuse/abuse these components or trigger unexpected behavior on the blockchain due to logic bugs or missing checks. Since the Frontier codebase is entirely open source, it is realistic that an adversary could analyze the source code while preparing an attack.

Fuzz testing is a technique to identify issues in code that handles untrusted input. In Frontier's case these are extrinsics in the runtime and smart contracts deployed on Frontier's EVM. Fuzz testing works by taking some valid input for a method under test, applying a semi-random mutation to it, and then invoking the method under test again with this semi-valid input. Through repeating this process, fuzz testing can unearth inputs that would cause a crash or other undefined behavior (e.g., integer overflows) in the method under test. SRLabs implemented two fuzzing harnesses to test both the EVM and its integration into Frontier through pallet-evm. The fuzzing harnesses utilized several invariants to thoroughly verify that the intended functionality is accurately implemented.

4.4 Remediation support

The final step is supporting Frontier with the remediation process of the identified issues. Each finding was documented and published with mitigation recommendations. Once the mitigation

solution is implemented, the fix is verified by us to ensure that it mitigates the issue and does not introduce other bugs.

During the audit, findings were shared via a private GitHub repository. We also used a private Slack channel for asynchronous communication and status updates. In addition, biweekly jour fixe meetings were held to provide detailed updates and address open questions.

5 Static analysis assessment

Throughout our auditing process, we utilized static analysis in our workflow to identify rule-based vulnerabilities in the EVM and Frontier codebase. Semgrep [4] and Dylint [5] were the primary tools utilized; with general rules for Rust such as unsafe arithmetic and integer truncation. Our static analysis yielded several findings.

We found 77 potential panic locations in Frontier, 21 integer coercions, and 20 unsafe math operations. In the EVM, we found 25 integer coercions and 86 unsafe math operations. There are seldom if any scenarios where blind unwraps, as coercions and unsafe math are encouraged, and we strongly recommend migrating to safer primitives such as checked or saturating math and error propagation.

We additionally ran cargo-audit for Frontier, an automated method to detect dependencies with known vulnerabilities, which yielded 16 results, including 6 vulnerabilities and 8 warnings. The results are shown in Table 6. Based on our assessment, these findings do not affect Frontier directly. Nonetheless, we advise applying the most recent update to guarantee comprehensive protection.

Dependency	Version	Description	Type
curve25519-dalek	3.2.0,	Timing variability in curve25519-dalek's `Scalar29::sub/Scalar52::sub.`	Vulnerability
Rustls	0.20.9	`rustls::ConnectionCommon::complete_io` could fall into an infinite loop based on network input.	Vulnerability
Idna	0.2.3, 0.4.0	Idna accepts Punycode labels that do not produce any non-ASCII when decoded.	Vulnerability
Openssl	0.10.71	Use-After-Free in `Md::fetch` and `Cipher::fetch`	Vulnerability
Sqlx	0.7.4	Binary protocol misinterpretation caused by overflowing casts	Vulnerability
Ring	0.16.20, 0.17.8	Some AES functions may panic when overflow checking is enabled	Vulnerability
Derivative	2.2.0	derivative is unmaintained.	Warning
Instant	0.1.13	instant is unmaintained.	Warning
Mach	0.3.2	mach is unmaintained.	Warning
parity-wasm	0.45.0	Crate parity-wasm is deprecated by the author.	Warning
proc-macro-error	1.0.4	proc-macro-error is unmaintained.	Warning
ring	0.16.20	Versions of ring prior to 0.17 are unmaintained.	Warning
trust-dns-proto	0.23.2	trust-dns-proto is unmaintained.	Warning
Paste	1.0.15	Paste is unmaintained.	Warning

Table 6: Cargo Audit findings for Frontier

6 Dynamic analysis assessment

During the audit, we utilized fuzz testing to identify bugs in Frontier and EVM. By applying fuzz testing, we aim to uncover issues that may go undetected through manual code review. For this purpose, we created two harnesses. One for the frontier template and one for the EVM implementation. The frontier template harness targeted the integration of the EVM into polkadot-sdk and the EVM harness focused on the EVM interpreter. The fuzzing campaign utilized a grammar-aware approach, where the fuzzer semantically mutated and generated EVM bytecode and contract call data.

Both harnesses were equipped with a variety of invariants, considering PoV (Proof of Validity) size, storage growth ratios, gas limits, gas consumption and bounded execution times.

To extend our dynamic analysis assessment, we thoroughly evaluated individual EVM opcodes and fuzzer generated sequences of EVM opcodes for any pricing concerns, factoring in memory usage, execution time and gas consumption.

Fuzzing campaign orchestration: We chose Ziggy [6], an open-source tool developed in-house, as our fuzzing orchestration tool. Ziggy uses two state of the art fuzzers, AFL++ [7] and honggfuzz [8] under the hood.

Coverage analysis and optimization: Although modern fuzzers can achieve good coverage by utilizing various techniques, we manually generated some seeds to target specific functionalities that were not covered by the fuzzer after a certain period. This approach assisted the fuzzer and optimized the overall coverage, ensuring more comprehensive testing.

Our dynamic analysis assessment yielded no findings.

6.1 Fuzzing Campaign Coverage

In our coverage analysis in Table 7, we only measure the Frontier codebase coverage that the harness is targeting.

Component	Repository	Coverage achieved
EVM	EVM [3]	92%
pallet-evm	Frontier [2]	75%

Table 7: Coverage analysis

Despite high coverage, there were no findings. SRLabs is in talks to integrate the harnesses in the Frontier and EVM repositories to allow for continuous fuzzing of both codebases.

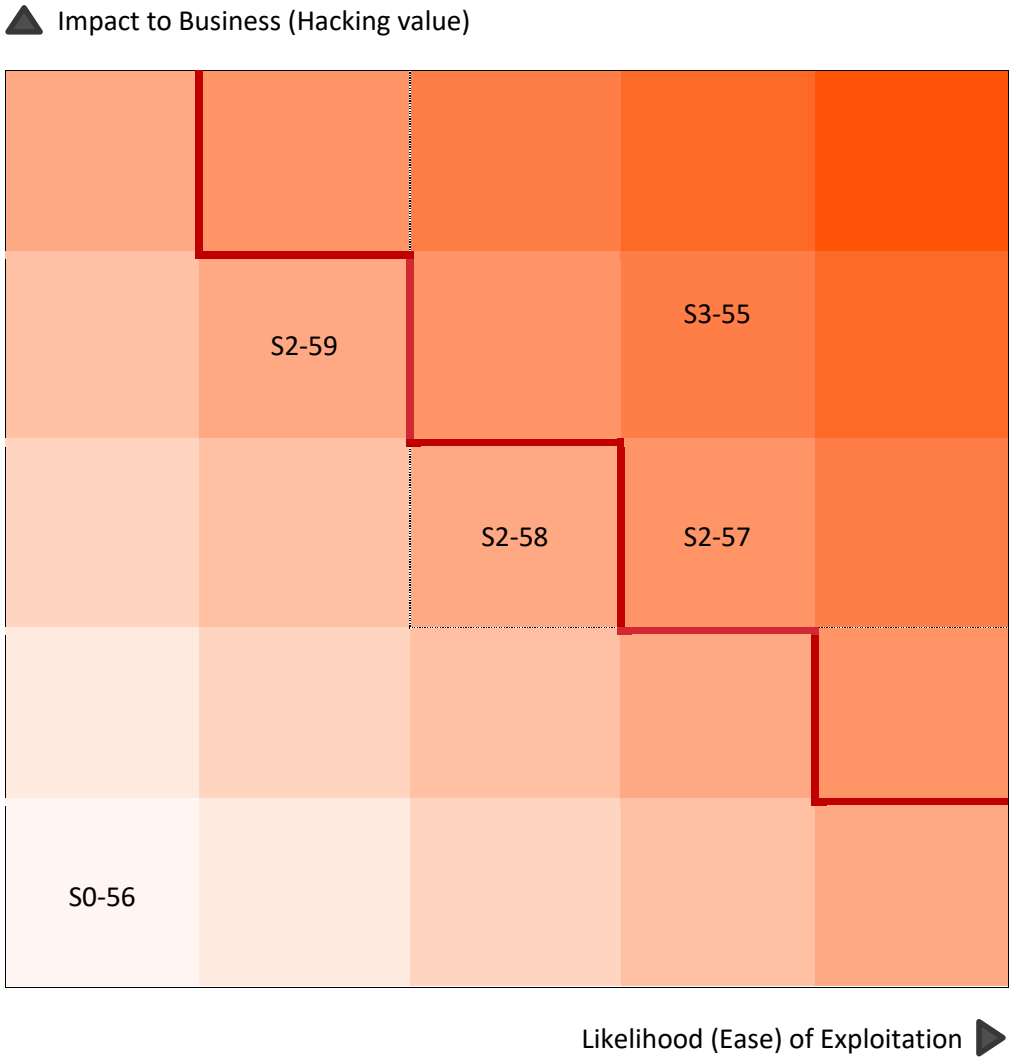
7 Findings summary

We identified 5 issues during our analysis of the various modules in scope in the Frontier codebase that enabled some of the attacks outlined above. In summary, we found 1 high-severity, 3 medium-severity and 1 information-level issues. An overview of all findings can be found in Table 8.

High	1
Medium	3
Low	0
Informational	1
Total Issues	**5**

7.1 Risk profile

The chart below summarizes vulnerabilities according to business impact and likelihood of exploitation, increasing to the top right. The red margin separates the high-critical issues from medium/low/informational ones.



7.2 Issue summary

ID	Issue	Severity	Status
S3-55 [9]	Constructing smart contract can bypass precompile checks	High	Mitigated [10]
S2-59 [11]	Missing inherent check can cause artificial gas price inflation	Medium	Acknowledged
S2-58 [12]	Silent failure in `Curve25519` arithmetic precompiles	Medium	Acknowledged
S2-57 [13]	Various underpriced precompiles can lead to DoS attack	Medium	Acknowledged
S0-56 [14]	FeeMultiplierUpdate not set to a TargetedFeeAdjustment type	Info	Acknowledged

Table 8: Findings overview

8 Detailed findings

8.1 S3-55: Constructing smart contract can bypass precompile address bounding

Attack scenario	**The attack bypasses precompile call filtering by performing calls from a constructing smart contract**
Component	/precompiles/src/precompile_set.rs
Tracking	https://github.com/moonbeam-foundation/sr-moonbeam/issues/55
Attack impact	Attackers may execute complex logic-based attacks involving multiple calls, if the precompile is assumed to be safe from contract calls
Severity	High
Status	Mitigated [10]

Background
There are various account address types in Frontier, e.g. precompiled contracts, smart contracts, and externally owned accounts. Some EVM mechanisms should be unreachable by certain types of accounts for safety.

For precompiles to be callable by smart contracts they must be explicitly configured as `CallableByContract` (for example here in the Moonbeam configuration [15]). If this configuration is absent, then the precompile should be unreachable via smart contract accounts.

Issue description
To implement caller bounding, in precompile_set.rs the type of calling address is calculated by `is_address_eoa_or_precompile()` [16] which calls into `get_address_type()` matching the returned account type.

In get_address_type() [17] the following check is performed:

```
if code_len == 0 {
    return Ok(AddressType::EOA);
}
```

This check falsely assumes it is impossible for a smart contract account to have a `code_len` of zero. This assumption does not hold true for contracts under construction, since the runtime bytecode in the create transaction is not fully loaded into storage.

Smart contract bytecode for create transactions are separated into initialization bytecode and runtime bytecode. The initialization bytecode is responsible for handling constructor logic and loading the runtime bytecode into storage, during this process external calls within the constructor will result in `msg.sender` having a `code_len` of zero.

If the check is bypassed, `common_checks()` [18] will fail to revert with a check against the config (`CallableByContract`) and the address type is returned.

We compiled a moonbeam collator node, with `CallableByContract` disabled on the author mapping precompile. This means that all calls from a smart contract should be filtered by AuthorMapping.

We then deployed a smart contract and executed a call to the address of the AuthorMapping precompile `0x00...807` within the constructor. This call succeeded because at construction the contract has a zero code size.

Finally, to test that we actually revert in a normal call from a fully initialized contract: we executed a secondary call, this time from the fully deployed smart contract's function to the same precompile.

As expected the secondary call failed since the smart contract was fully deployed, the following reversion message was returned indicating the exact check that the first call bypassed.

```
VM Exception while processing transaction: revert Function not callable
by smart contract
```

This error message indicates that under conventional circumstances the call is correctly reverted, however from constructing smart contracts, it may be bypassed.

Risk
This is an incomplete or inadequate core security feature that fails to properly handle all circumstances where the calling account is a smart contract.

This risk, in isolation, to Frontier or Moonbeam is very low, since there are no precompiles that currently rely on not being reachable by smart contracts, however for third-party developers the risk may be substantially higher.

Developers may place too much trust on the correctness of this call filter, and as a result open themselves to unique abuse cases originating from smart contracts.

Mitigation
We recommend performing this check based on transaction properties. For example, if the origin of a transaction `tx.origin` is equal to the `caller` (`msg.sender`) the call will never have originated from a smart contract.

8.2 S2-59: Missing check_inherent for note_min_gas_price_target inflates gas price

Attack scenario	**Block authors can dishonestly raise gas prices at block production with zero disincentives**
Component	/frame/dynamic-fee/src/lib.rs
Tracking	https://github.com/moonbeam-foundation/sr-moonbeam/issues/59
Attack impact	The gas price will be artificially inflated for all users irrespective of genuine network usage, potentially causing a denial-of-service attack
Severity	Medium
Status	Acknowledged

Background
In the Polkadot-SDK, inherent extrinsics are unsigned transactions added by block producers to include local, node-specific data directly into blocks. Furthermore, gas is the price paid in the native currency for each unit of gas used.

Issue description
The extrinsic `note_min_gas_price_target` is an inherent extrinsic, meaning only the block producer can call it. To ensure correctness, the `ProvideInherent` trait should be implemented for each inherent, which includes the `check_inherent` call. This allows other nodes to verify if the input (in this case, the `target` value) is correct.

However, the `check_inherent` function has not been implemented for `note_min_gas_price_target`. This lets the block producer set the target value without verification. The target is then used to set the `MinGasPrice`, which has an upper and lower bound defined in the `on_initialize` hook. The block producer can set the target to the upper bound. Which also increases the upper and lower bounds for the next block. Over time, this could result in continuously raising the gas price, making contract execution too expensive and ineffective for users.

Risk
An attacker could use this flaw to manipulate the gas price, potentially leading to significantly inflated transaction fees. Such manipulation could render contract execution prohibitively expensive for users, effectively resulting in a denial-of-service condition for the network.

Mitigation
Implement the `check_inherent` function to verify that the target price is agreed upon by all the nodes in the network.

8.3 S2-58: Silent failure in Curve25519 arithmetic precompiles with malformed points

Attack scenario	**An attacker submits an invalid Ristretto point causing the contract to treat it as the identity element**
Component	/frame/evm/precompile/curve25519/src/lib.rs
Tracking	https://github.com/moonbeam-foundation/sr-moonbeam/issues/58
Attack impact	An attacker may abuse this to bypasses cryptographic checks, enabling signature forgery or compromising key exchanges
Severity	Medium
Status	Acknowledged

Background
The Curve25519 arithmetic precompiles allow smart contract developers to cheaply utilize these curve operations in their smart contracts.

Issue description
The `Curve25519Add` and `Curve25519ScalarMul` precompiles incorrectly handle invalid Ristretto point representations. Instead of returning an error, they silently treat invalid input bytes as the Ristretto identity element, leading to potentially incorrect cryptographic results.

In `/frame/evm/precompile/curve25519/src/lib.rs`, when processing input points for both addition and scalar multiplication, in the execute function, the code attempts to decompress the 32-byte input using `point.decompress()`.

If the input bytes do not represent a valid compressed Ristretto point, `decompress()` returns `None`. However, the code uses `unwrap_or_else(RistrettoPoint::identity)`, which replaces this `None` result with the `RistrettoPoint::identity()` element without signalling any error to the caller.

Risk
Silently treating invalid cryptographic points as the identity element could introduce critical vulnerabilities in smart contracts. This behavior could compromise confidentiality in key exchanges or allow threshold signature requirements to be bypassed. For example, in a multi-signature scheme, an attacker could submit an invalid compressed Ristretto point as a public key. Treated as the identity element, it would be incorrectly counted toward the signing threshold, enabling signature forgery with fewer valid participants.

Mitigation
We recommend modifying the execute function in both `Curve25519Add` and `Curve25519ScalarMul`. Instead of using `unwrap_or_else(RistrettoPoint::identity)`, check the result of `point.decompress()`. If it returns `None`, the function should immediately return a `PrecompileFailure::Error` indicating invalid point data, rather than proceeding with the identity element.

8.4 S2-57: Various underpriced precompiles can lead to DoS attack

Attack scenario	**An attacker may exploit underpriced precompiles to consume excessive CPU resources on the network**
Component	/frame/evm/precompile/*
Tracking	https://github.com/moonbeam-foundation/sr-moonbeam/issues/57
Attack impact	An attacker may halt or delay block production while paying minimal fees
Severity	Medium
Status	Acknowledged

Background
Both `Curve25519Add` and `Sha3FIPS512` precompiles implement the `LinearCostPrecompile`, which means that their gas usage is dependent on the input data size, a factor rounded up to the nearest number of words – as stated in the Ethereum yellow paper [19] for SHA2-256 and RIPEMD-160.

Issue description
The gas cost for the `Curve25519Add` precompile is under-priced relative to its compute, specifically when compared to the `Sha3FIPS512` precompile. Both use the same gas calculation constants (see here [20] [21]), but `Curve25519Add` requires significantly more computation.

We designed several test cases, detailed directly to the Frontier development team, which is out-of-scope in this report. During these benchmarks, we noticed a discrepancy between the gas-cost and CPU-cost ratio of `Sha3FIPS512` and `Curve25519Add`.

The test results demonstrate a significant disparity in execution time:

- Sha3FIPS512: gas = **84**, time = **1.333µs**
- Curve25519Add: gas = **84**, time = **12.416µs**

Both precompiles consume the same amount of gas, but one requires around ten times higher amount of compute.

Since both use the same parameters for the linear cost calculation (`BASE=60` and `WORD=10`), for the same input length, the gas record should be identical.

Furthermore, we identified that the `Curve25519ScalarMul` was also underpriced. The following are our results from this additional analysis:

- `Sha3FIPS512`: gas = **84**, time = **2.041µs**
- `Curve25519Add`: gas = **84**, time = **10.333µs**
- `Curve25519ScalarMul`: gas = **84**, time = **37.75µs**

Risk
Attackers can craft transactions calling the underpriced precompile, consuming substantial node resources (CPU time) while paying relatively little gas, potentially leading to availability and denial-of-service issues which slows down block processing.

Mitigation
We recommend reviewing and improving the benchmarking process of every precompile that is a `LinearCostPrecompile`. Especially, for precompiles that are not native to Ethereum such as `Curve25519Add`, the `BASE` and `WORD` variables should be benchmarked appropriately.

8.5 S0-56: FeeMultiplierUpdate not set to a TargetedFeeAdjustment type

Attack scenario	**Developers building based on the template my not understand the security implications of this configuration**
Component	/template/runtime/src/lib.rs
Tracking	https://github.com/moonbeam-foundation/sr-moonbeam/issues/56
Attack impact	Attackers may abuse constant target fees to censor zero-tip transactions while paying only the regular transaction fee
Severity	Info
Status	Acknowledged

Background
Frontier provides a template for developers to get started using Frontier. In that template, the network fees will not automatically adjust if the network becomes overloaded with transactions, because the runtime does not set `FeeMultiplierUpdate` to a `TargetedFeeAdjustment`.

Issue description
The runtime configuration sets `FeeMultiplierUpdate` to `ConstFeeMultiplier` [22] which does not take into consideration the current usage of the network.

```
parameter_types! {
    pub FeeMultiplier: Multiplier = Multiplier::one();
}

impl pallet_transaction_payment::Config for Runtime {
    … … …
    type FeeMultiplierUpdate = ConstFeeMultiplier<FeeMultiplier>;
}
```

Risk
Since the fees do not reflect congestion, an attacker could cause a denial-of-service against all zero-tip transactions for an extended period by just paying the regular transaction fees (and not exponentially raising fees, which would at least limit the duration of that attack).

Mitigation
We recommended setting `FeeMultiplierUpdate` to a `TargetedFeeAdjustment` type, an example of such implementation may be found in following configuration [23], where `SlowAdjustingFeeUpdate` is used.

9 Bibliography

[1] [Online]. Available: https://github.com/srlabs/substrate-runtime-fuzzer.

[2] [Online]. Available: https://github.com/polkadot-evm/frontier.

[3] [Online]. Available: https://github.com/rust-ethereum/evm.

[4] [Online]. Available: https://semgrep.dev/.

[5] [Online]. Available: https://github.com/trailofbits/dylint.

[6] [Online]. Available: https://github.com/srlabs/ziggy.

[7] [Online]. Available: https://github.com/AFLplusplus/AFLplusplus.

[8] [Online]. Available: https://github.com/google/honggfuzz.

[9] [Online]. Available: https://github.com/moonbeam-foundation/sr-moonbeam/issues/55.

[10] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/pull/3273.

[11] [Online]. Available: https://github.com/moonbeam-foundation/sr-moonbeam/issues/59.

[12] [Online]. Available: https://github.com/moonbeam-foundation/sr-moonbeam/issues/58.

[13] [Online]. Available: https://github.com/moonbeam-foundation/sr-moonbeam/issues/57.

[14] [Online]. Available: https://github.com/moonbeam-foundation/sr-moonbeam/issues/56.

[15] [Online]. Available: https://github.com/moonbeam-foundation/moonbeam/blob/550a7f66351d71f3c4ec1fbc1de65545ba763cc6/runtime/moonbase/src/precompiles.rs#L114.

[16] [Online]. Available: https://github.com/polkadot-evm/frontier/blob/c0d5cb8b08c548e747b911fdeb46e801e5aa4c04/precompiles/src/precompile_set.rs#L348.

[17] [Online]. Available: https://github.com/polkadot-evm/frontier/blob/c0d5cb8b08c548e747b911fdeb46e801e5aa4c04/precompiles/src/precompile_set.rs#L330.

[18] [Online]. Available: https://github.com/polkadot-evm/frontier/blob/c0d5cb8b08c548e747b911fdeb46e801e5aa4c04/precompiles/src/precompile_set.rs#L382.

[19] [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.

[20] [Online]. Available: (https://github.com/polkadot-evm/frontier/blob/c0d5cb8b08c548e747b911fdeb46e801e5aa4c04/frame/evm/precompile/curve25519/src/lib.rs#L35-L36.

[21] [Online]. Available: https://github.com/polkadot-evm/frontier/blob/c0d5cb8b08c548e747b911fdeb46e801e5aa4c04/frame/evm/precompile/sha3fips/src/lib.rs#L46-L47.

[22] [Online]. Available: https://github.com/polkadot-evm/frontier/blob/d5755b5d0cdc84cda71bd267c761b72e54ca7650/template/runtime/src/lib.rs#L306.

[23] [Online]. Available: https://github.com/paritytech/polkadot-sdk/blob/18db502172bdf438f086cd5964c646b318b8ad37/polkadot/runtime/common/src/lib.rs#L110.

[24] [Online]. Available: https://github.com/polkadot-evm/frontier/blob/c0d5cb8b08c548e747b911fdeb46e801e5aa4c04/precompiles/src/precompile_set.rs#L330.

[25] [Online]. Available: https://github.com/polkadot-evm/frontier/tree/master/template/runtime.

Appendix A: Technical services

Security Research Labs delivers extensive technical expertise to meet your security needs. Our comprehensive services include software and hardware evaluation, penetration testing, red team testing, incident response, and reverse engineering. We aim to equip your organization with the security knowledge essential for achieving your objectives.

SOFTWARE EVALUATION We provide assessments of application, system, and mobile code, drawing on our employees' decades of experience in developing and securing a wide variety of applications. Our work includes design and architecture reviews, data flow and threat modelling, and code analysis with targeted fuzzing to find exploitable issues.

BLOCKCHAIN SECURITY ASSESSMENTS We offer specialized security assessments for blockchain technologies, focusing on the unique challenges posed by decentralized systems. Our services include smart contract audits, consensus mechanism evaluations, and vulnerability assessments specific to blockchain infrastructure. Leveraging our deep understanding of blockchain technology, we ensure your decentralized applications and networks are secure and robust.

POLKADOT ECOSYSTEM SECURITY We provide comprehensive security services tailored to the Polkadot ecosystem, including parachains, relay chains, and cross-chain communication protocols. Our expertise covers runtime misconfiguration detection, benchmarking validation, cryptographic implementation reviews, and XCM exploitation prevention. Our goal is to help you maintain a secure and resilient Polkadot environment, safeguarding your network against potential threats.

TELCO SECURITY We deliver specialized security assessments for telecommunications networks, addressing the unique challenges of securing large-scale and critical communication infrastructures. Our services encompass vulnerability assessments, secure network architecture reviews, and protocol analysis. With a deep understanding of telco environments, we ensure robust protection against cyberthreats, helping maintain the integrity and availability of your telecommunications services.

DEVICE TESTING Our comprehensive device testing services cover a wide range of hardware, from IoT devices and embedded systems to consumer electronics and industrial controls. We perform rigorous security evaluations, including firmware analysis, penetration testing, and hardware-level assessments, to identify vulnerabilities and ensure your devices meet the highest security standards. Our goal is to safeguard your hardware against potential attacks and operational failures.

CODE AUDITING We provide in-depth code auditing services to identify and mitigate security vulnerabilities within your software. Our approach includes thorough manual reviews, automated static analysis, and targeted fuzzing to uncover critical issues such as logic flaws, insecure coding practices, and exploitable vulnerabilities. By leveraging our expertise in secure software development, we help you enhance the security and reliability of your codebase, ensuring robust protection against potential threats.

PENETRATION & RED TEAM TESTING We perform high-end penetration tests that mimic the work of sophisticated adversaries. We follow a formal penetration testing methodology that emphasizes repeatable, actionable results that give your team a sense of the overall security posture of your organization.

SOURCE CODE-ASSISTED SECURITY EVALUATIONS We conduct security evaluations and penetration tests based on our code-assisted methodology that lets us find deeper vulnerabilities, logic flaws,

and fuzzing targets than a black-box test would reveal. This gives your team a stronger assurance that the significant security-impacting flaws have been found and corrected.

SECURITY DEVELOPMENT LIFECYCLE CONSULTING We guide organizations through the Security Development Lifecycle to integrate security at every phase of software development. Our services include secure coding training, threat moelling, security design reviews, and automated security testing implementation. By embedding security practices into your development processes, we help you proactively identify and mitigate vulnerabilities, ensuring robust and secure software delivery from inception to deployment.

REVERSE ENGINEERING We assist clients with reverse engineering efforts that are not associated with malware or incident response. We also provide expertise in investigations and litigation by acting as experts in cases of suspected intellectual property theft.

HARDWARE EVALUATION We evaluate new hardware devices ranging from novel microprocessor designs, embedded systems, mobile devices, and consumer-facing end products to core networking equipment that powers Internet backbones.

VULNERABILITY PRIORITIZATION We streamline vulnerability information processing by consolidating data from compliance checks, audit findings, penetration tests, and red team insights. Our prioritization and automation strategies ensure that the most critical vulnerabilities are addressed promptly, enhancing your organization's security posture. By systematically categorizing and prioritizing risks, we help you focus on the most impactful threats, ensuring efficient and effective remediation efforts.

SECURITY MATURITY REVIEW We conduct comprehensive security maturity reviews to evaluate your organization's current security practices and identify areas for improvement. Our assessments cover a wide range of criteria, including policy development, risk management, incident response, and security awareness. By benchmarking against industry standards and best practices, we provide actionable insights and recommendations to enhance your overall security posture and guide your organization toward achieving higher levels of security maturity.

SECURITY TEAM INCUBATION We provide comprehensive support for building security teams for new, large-scale IT ventures. From Day 1, our ramp-up program offers essential security advisory and assurance, helping you establish a robust security foundation. With our proven track record in securing billion-dollar investments and launching secure telco networks globally, we ensure your new enterprise is protected against cyberthreats from the start.

HACKING INCIDENT SUPPORT We offer immediate and comprehensive support in the event of a hacking incident, providing expert analysis, containment, and remediation. Our services include detailed forensics, malware analysis, and root cause determination, along with actionable recommendations to prevent future incidents. With our rapid response and deep expertise, we help you mitigate damage, recover swiftly, and strengthen your defenses against potential threats.