TON Blockchain TVM Upgrade

Security Assessment

October 27, 2023

Prepared for:
Dr. Elias
TON Foundation

Prepared by: **Samuel Moelius, Evan Sultanik, and Henrik Brodin**

About Trail of Bits

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

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Samuel Moelius, Consultant
samuel.moelius@trailofbits.com

Evan Sultanik, Consultant
evan.sultanik@trailofbits.com

Henrik Brodin, Consultant
henrik.brodin@trailofbits.com

Project Timeline

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

Date	Event
August 3, 2023	Pre-project kickoff call
August 14, 2023	Status update meeting #1 (canceled)
August 21, 2023	Status update meeting #2
August 28, 2023	Delivery of report draft
August 28, 2023	Report readout meeting
September 26, 2023	Delivery of comprehensive report
October 23, 2023	Fix review kickoff
October 27, 2023	Delivery of comprehensive report with fix review

Executive Summary

Engagement Overview

TON Foundation engaged Trail of Bits to review the security of its upgrade to the TON Virtual Machine (TVM). The upgrade changes the way that various aspects of the TVM work and introduces many new instructions. Many of the new instructions add support for new hashing and cryptographic algorithms.

A team of three consultants conducted the review from August 7 to August 25, 2023, for a total of seven engineer-weeks of effort. Our testing efforts focused on code that was added or changed by the upgrade. With full access to the source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes.

Observations and Impact

The two areas we find most concerning involve the project's integer type and its use of tests.

- `BigInt` is used to perform arithmetic on integers larger than the host machine's native word size. For reasons that are not clear to us, it allows for multiple representations of the same integer. A `normalize` function can be used to put an integer into a canonical form. However, it is similarly unclear to us when such calls are needed. That is, while calls to `normalize` appear in various parts of the code, we could not infer a pattern. Two high-severity findings in this report involve missing calls to `normalize` (TOB-TVMUP-11 and TOB-TVMUP-15).

  Arithmetic is an area of special concern to blockchain applications, as many such applications maintain account balances. Generally speaking, a flaw that causes an account's balance to be computed incorrectly could be exploited.

  For the reasons just given, blockchain code involving arithmetic should be straightforward, well documented, and written in a way that makes the absence of errors obvious. For reasons outlined above, `BigInt`'s current implementation does not seem to meet these requirements.

- Many of the problems exposed in this report could have been found through better testing. The project does use the `CTest` framework, but the extent is unclear. For example, there is no evidence that the tests are run in CI. Furthermore, some tests that did not pass were referred to as "old." Thus, testing does not seem to be applied rigorously and effectively.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that TON Foundation 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.

- **Adopt an integer type that allows for only one representation of each integer.** Such an integer type could be external (e.g., the one provided by the GMP library), or it could be an adaptation of the existing `BigInt` implementation. Regardless, the current `BigInt` implementation seems highly error-prone, and an alternative should be sought.

- **Expand the project's use of tests.** Many of the problems exposed in this report could have been found through better testing. Specifically, the following steps should be taken.

    - Add instructions to the project's `README.md` file on how to run the tests.

    - Run the tests in CI.

    - Regularly compute and review test coverage to ensure that important conditions are tested.

    - Explore advanced testing methods such as property-based and fuzz testing.

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

EXPOSURE ANALYSIS

Severity	*Count*
High	6
Medium	0
Low	4
Informational	9
Undetermined	3

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	1
Data Validation	5
Denial of Service	1
Patching	3
Testing	1
Timing	1
Undefined Behavior	10

After the engagement concluded, Trail of Bits reviewed the fixes and mitigations implemented by the TON team for the issues identified in this report. The following table summarizes the results of our fix review. For more information, refer to the detailed fix review results in appendix G.

FIX STATUSES

Fix Status	*Count*
Resolved	16
Partially Resolved	5
Undetermined	1

Project Goals

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

- Are the new opcodes implemented correctly?
- Are newly introduced dependencies (e.g., `libsodium` and `blst`) used correctly?
- Are the new opcodes' behaviors consistent with the documentation?
- Is there any way that newly introduced functionality could cause users to lose money?
- Does any of the newly introduced code contain undefined behavior?

Project Targets

The engagement involved a review and testing of the following target.

TON

Repository	https://github.com/ton-blockchain/ton
Version	`6074702d059fee2b9456e47c294693447ca222ef`
Type	C++
Platform	POSIX

TON Documentation

Repository	https://github.com/ton-community/ton-docs
Version	`8b1140e38a1b148498706c9375eb34034f2967d1`
Type	Documentation

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:

- **Documentation review:** We carefully reviewed the TVM upgrade documentation.
- **Static analysis:** We ran the static analysis tools Cppcheck, Clang Static Analyzer, and CodeQL over the codebase and reviewed their results.
- **Test coverage review:** We ran all tests and reviewed which did and did not pass. We also computed the tests' coverage using `gcovr` and reviewed the results to look for important conditions that could be missed.
- **Fuzzing:** We differentially fuzzed the TVM's Keccak implementation against a similar, third-party implementation. We also fuzzed the new arithmetic instructions against manual computations written from the instructions' specifications.
- **Manual review:** We manually reviewed all code involved in the upgrade (i.e., PR #686), but with special focus on the following areas:
    - Changes affecting the TVM's behavior generally (e.g., changes to the c7 tuple)
    - Changes to `transaction.cpp` enabled by setting the `global_version` to be at least 4
    - New arithmetic instructions
    - New hashing instructions
    - New hashing implementations (Keccak-256 and Keccak-512)
    - secp256k1 and secp256r1 instructions
    - Ristretto instructions
    - BLS12-381 instructions
    - The `RUNVM` instruction

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- As mentioned elsewhere, the state of the project's tests is unclear. We commonly use tests to learn how functions are meant to be used and what invariants they are expected to maintain. Our ability to do this with the current tests was limited.

- Due to time constraints, the Keccak implementations were fuzzed, but not the instructions that exercise the Keccak implementations.

- We were not able to exhaustively check all instructions' gas costs. We recommend that gas costs be verified with tests.

Automated Testing

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

Test Harness Configuration

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

- Cppcheck: A static analysis tool for C/C++ code that focuses on detecting undefined behavior and risky coding constructs
- Clang Static Analyzer: A source code analysis tool that finds bugs in C, C++, and Objective-C programs
- CodeQL: A semantic code analysis engine that allows users to query code as though it were data
    - We ran CodeQL with both the publicly available queries and a collection of our own internally developed queries.
- AFLplusplus: A widely used fork of Google's American Fuzzy Lop (AFL) fuzzer

Fuzzing

We developed fuzzers for the targets listed in the table below.

The upgrade to the TVM introduced new hashing modes, including Keccak-256 and Keccak-512. We differentially fuzzed the implementations of these new hashing modes against slight variants of the SHA-256 and SHA-512 reference implementations. These efforts resulted in one informational-severity finding (TOB-TVMUP-3). The code used to perform the fuzzing appears in appendix E.

We also fuzzed 24 new arithmetic instructions. Specifically, we wrote code to compute each instruction’s result based on its specification. We then called the instruction and verified that the expected result matched the one that was actually computed. These efforts resulted in one low-severity finding (TOB-TVMUP-10) and two high-severity findings (TOB-TVMUP-11 and TOB-TVMUP-15). The code used to perform the fuzzing appears in appendix F.

Fuzz Targets	Findings
`Hasher::KECCAK256` and `Hasher::KECCAK512`	TOB-TVMUP-3
The 24 new arithmetic instructions described in the upgrade documentation, where ? determines the rounding mode and ranges over the empty string (floor), R (round), and C (ceiling) • `MULADDDIVMOD?` • `ADDDIVMOD?` • `ADDRSHIFTMOD?` • `z ADDRSHIFT?#MOD` • `MULADDRSHIFT?MOD` • `z MULADDRSHIFT?#MOD` • `LSHIFTADDDIVMOD?` • `y LSHIFT#ADDDIVMOD?`	TOB-TVMUP-10 TOB-TVMUP-11 TOB-TVMUP-15

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	Potentially overflowing arithmetic is not sufficiently documented. No explicit testing strategy has been identified to increase confidence in the system's arithmetic. The test suite does not cover several arithmetic edge cases.	**Weak**
Auditing	The TVM has robust logging and debugging capabilities.	**Satisfactory**
Authentication / Access Controls	Authentication and access controls were not within the scope of this audit.	**Not Considered**
Complexity Management	We found the code to be relatively clear. The constructs and idioms used are well known, and the code generally contains few surprises. Nonetheless, we found the code to be lacking comments. Additional comments could be used to clarify each function's purpose and to communicate the developer's intent in various parts of the code. Also, the code relies on outdated dependencies.	**Moderate**
Cryptography and Key Management	We found only minor issues related to cryptography in newly introduced instructions. However, the code uses deprecated OpenSSL APIs, which should be updated. The code also contains implementations of deprecated cryptographic algorithms, which should be removed.	**Moderate**
Decentralization	Decentralization was not within the scope of this audit.	**Not Considered**
Documentation	TON has comprehensive and thorough documentation. However, there appear to be different versions in	**Moderate**

	existence (e.g., the version provided to us versus a version available on ton.docs.org). Moreover, we found several discrepancies between the documentation and the implementation. Also, as mentioned elsewhere, the code would benefit from additional inline comments.	
Front-Running Resistance	Some TVM changes did affect how transactions are handled. These changes did not have immediate or obvious impacts on front-running resistance. However, further investigation would be required to reach a meaningful conclusion.	**Further Investigation Required**
Low-Level Manipulation	The code's big integer type allows for multiple representations of the same integer. A `normalize` function can be used to put an integer into a canonical form. However, it is not clear when the function must be used. We found multiple bugs that seem to result from failures to call `normalize`.	**Weak**
Testing and Verification	The project lacks instructions for building and running its tests. There is no evidence that tests are run in CI. Some critical pieces of code (e.g., `transaction.cpp`) appear to be untested. Also, many of the problems exposed in this report could have been found through better testing.	**Weak**

Summary of Findings

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

ID	Title	Type	Severity
1	Inadequate testing	Testing	**Informational**
2	Insufficient code comments	Patching	**Informational**
3	Hash bit ordering differs from FIPS 202	Cryptography	**Informational**
4	Action phase fines can be bypassed	Data Validation	**Undetermined**
5	Use of deprecated OpenSSL APIs	Patching	**Informational**
6	MULADDDIVMOD and related instructions have unclear behavior	Undefined Behavior	**Informational**
7	Undefined behavior in CyclicBlobViewImpl	Undefined Behavior	**High**
8	Use of blst version with new-delete mismatch	Undefined Behavior	**High**
9	Arithmetic opcodes handled inconsistently	Undefined Behavior	**Informational**
10	Inconsistencies between arithmetic operations' implementation and specification	Undefined Behavior	**Low**
11	Missing call to normalize in ADDDIVMOD implementation	Undefined Behavior	**High**
12	Use of deprecated cryptographic APIs	Patching	**Informational**

13	Bignum can segfault when converting to string or hex	Data Validation	**Informational**
14	Risk of infinite loop during RaptorQ FEC	Denial of Service	**Undetermined**
15	Missing to call to normalize in MULADDRSHIFT#MOD implementation	Undefined Behavior	**High**
16	BLS gas costs are inconsistent with specification	Undefined Behavior	**Low**
17	Use of libsodium might stall the process	Timing	**Low**
18	RIST255_MUL uses nonstandard method for handling errors	Data Validation	**High**
19	Cell slices for public keys and signatures can have excess data	Data Validation	**Low**
20	Divergent behavior among BLS instructions when n is 0	Data Validation	**Informational**
21	Uninitialized data read when downcast_call fails	Undefined Behavior	**High**
22	Register c7 tuple element "previous blocks" can be null	Undefined Behavior	**Undetermined**

[1]

[1] For information about the fix statuses of these findings, refer to appendix G.

Detailed Findings

1. Inadequate testing	
Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-TVMUP-1
Target: `README.md` and the `.github/workflows` and `test` subdirectories	

Description
The TON repository is inadequately tested. Code should be tested thoroughly to help ensure its correctness.

The TON repository suffers from the following specific deficiencies:

- The repository does not contain instructions for running the tests.
- Several of the existing tests fail when run (figure 1.1).
- There is no evidence that tests are run in CI.
- Several source files appear to be completely untested (e.g., `transaction.cpp`, referenced in finding TOB-TVMUP-4).

```
$ make test
Running tests...
...
The following tests FAILED:
          4 - test-vm (Failed)
          7 - test-smartcont (Failed)
         11 - test-tonlib-offline (Failed)
Errors while running CTest
```

Figure 1.1: Output produced by running `make test`

Exploit Scenario
A bug is found in the TVM. The bug could have been exposed by more thorough unit tests.

Recommendations
Short term, take the following steps:

- Add instructions to the project's `README.md` file on how to run the tests.

- Run the tests in CI.
- Ensure that CI fails if any test fails.

Taking these steps will help increase confidence in the TVM.

Long term, regularly compute and review test coverage using a tool such as `gcovr`. Doing so will help ensure that the tests are relevant and that all important conditions are tested.

2. Insufficient code comments	
Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-TVMUP-2
Target: The `crypto` subdirectory	

Description

The TVM source code is inadequately commented. Having too few comments can cause code to be misunderstood, which increases the likelihood of an improper bugfix or a mis-implemented feature.

There are 138 files ending in `.cpp` in the `crypto` subdirectory. In total, those files consist of 97,649 lines. Of those lines, 4,015 contain line comments (i.e., match the regular expression `//.*`). Thus, approximately 4.11% of the lines composing all `.cpp` files in the `crypto` subdirectory contain line comments. (See appendix C for the raw data used for this calculation.)

Exploit Scenario

Alice, a TON developer, implements a new feature for the TVM. Alice misunderstands how the functions called by her new feature work. Alice introduces a vulnerability into the virtual machine as a result.

Recommendations

Short term, add comments to the source files in the `crypto` subdirectory. Ensure that, for each function accessible from outside its translation unit, at least one of the following is true.

- The function's definition is preceded by a comment.
- The function's prototype (in the function's respective header) is preceded by a comment.

Taking these steps will facilitate code review and reduce the likelihood that a developer introduces a bug into the code because of a misunderstanding.

Long term, regularly review code comments to ensure they are accurate. Documentation must be kept up to date to be beneficial.

3. Hash bit ordering differs from FIPS 202	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-TVMUP-3
Target: `crypto/vm/Hasher.cpp`, `crypto/common/bitstring.cpp`, TVM Instructions documentation	

Description

The TVM supports bit-level hashing, using most significant bit (MSB) ordering when the input does not end on a byte boundary. FIPS 202 defines the SHA3-256 and SHA3-512 algorithms, which are minor variants of algorithms used by TON. However, FIPS 202 uses least significant bit (LSB) ordering. This discrepancy could cause confusion for users.

The TVM treats byte sequences as having MSB ordering, generally.[2] This is evident from the TVM source code (figure 3.1).

```
int mask = (-0x100 >> bit_count) & (0xff >> to_offs);
```

Figure 3.1: `crypto/common/bitstring.cpp#L137`

However, FIPS 202 uses LSB ordering. This is clear from the specification[3] and from its reference implementation (figure 3.2).

```
if(csk->lastByteBitLen != 0)
    csk->lastByteValue = input[inputBitLen / 8] & ((1 << csk->lastByteBitLen) - 1);
/* strip unwanted bits */
```

Figure 3.2: `XKCP/lib/high/Keccak/SP800-185/SP800-185.inc#L88–L89`

As shown in figure 3.3, SHA3-256 and SHA3-512 (algorithms defined in FIPS 202) are only slight variants of Keccak-256 and Keccak-512 (algorithms that the TVM supports). Thus, users familiar with SHA3-256 and SHA3-512 are likely to assume that the TVM's Keccak-256 and Keccak-512 implementations use the same bit ordering.

```
/** Macro to initialize a SHA3-256 instance as specified in the FIPS 202 standard.
  */
#define Keccak_HashInitialize_SHA3_256(hashInstance)
Keccak_HashInitialize(hashInstance, 1088,  512, 256, 0x01)
```

[2] See section 1.0 (page 5) of the Telegram Open Network Virtual Machine white paper.
[3] See section B.1 (pages 26 and 27) of FIPS 202.

```
/** Macro to initialize a SHA3-384 instance as specified in the FIPS 202 standard.
  */
#define Keccak_HashInitialize_SHA3_384(hashInstance)
Keccak_HashInitialize(hashInstance,  832,  768, 384, 0x06)

/** Macro to initialize a SHA3-512 instance as specified in the FIPS 202 standard.
  */
#define Keccak_HashInitialize_SHA3_512(hashInstance)
Keccak_HashInitialize(hashInstance,  576, 1024, 512, 0x01)
```

Figure 3.3: Changes to `XKCP/lib/high/Keccak/FIPS202/KeccakHash.h#L71–L81` that cause it to implement Keccak-256 and Keccak-512

Exploit Scenario

Alice, a TON developer, writes code that applies Keccak-256 to sequences of partial bytes. Being familiar with SHA3-256, Alice expects bits from incomplete bytes to come from the lower end. Because the TVM instead takes the bits from the upper end, Alice's code does not work correctly.

Recommendations

Short term, conspicuously document this discrepancy in all TVM documentation involving hashing operations (e.g., under TVM Instructions). Doing so will help alert users to the fact that the TVM's behavior differs from certain related standards (e.g., FIPS 202).

Long term, as new instructions are introduced into the TVM, consider "prior art," that is, similar implementations that users may be familiar with. Where possible, emulate the existing behavior, or document the discrepancy. Doing so will reduce the likelihood of users being confused.

4. Action phase fines can be bypassed

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TVMUP-4
Target: `crypto/block/transaction.cpp`	

Description

Figure 4.1 is an excerpt of the new code responsible for calculating action phase fines.

```
1709     if (compute_phase) {
1710       new_funds -= compute_phase->gas_fees;
1711     }
1712     new_funds -= ap.action_fine;
1713     if (new_funds->sgn() < 0) {
1714       LOG(DEBUG)
1715           << "not enough value to transfer with the message: all of the inbound message value has been consumed";
1716       return skip_invalid ? 0 : 37;
1717     }
1718   }
1719   funds = std::min(funds, new_funds);
```

Figure 4.1: New fee and fine calculation code
(`ton/crypto/block/transaction.cpp#1709–1719`)

The `funds` and `new_funds` variables are signed integers, so they can go negative if the `gas_fees` and `action_fine` are sufficiently larger than the initial balance. Line 1716 handles such cases, rejecting the message. However, the transaction balance is not modified, since this would otherwise occur later in the function (line 1748 of figure 4.2).

```
1741   auto collect_fine = [&] {
1742     if (cfg.action_fine_enabled && !account.is_special) {
1743       td::uint64 fine = fine_per_cell * std::min<td::uint64>(max_cells, sstat.cells);
1744       if (ap.remaining_balance.grams->cmp(fine) < 0) {
1745         fine = ap.remaining_balance.grams->to_long();
1746       }
1747       ap.action_fine += fine;
1748       ap.remaining_balance.grams -= fine;
1749     }
1750   };
1751   if (sstat.cells > max_cells && max_cells < cfg.size_limits.max_msg_cells) {
1752     LOG(DEBUG) << "not enough funds to process a message (max_cells=" << max_cells << ")";
```

```
    collect_fine();
    return skip_invalid ? 0 : 40;
  }
```

Figure 4.2: The fine is subtracted from the balance (`crypto/block/transaction.cpp#1741–1755`)

This finding is of undetermined severity because there was insufficient time to produce a proof of concept demonstrating that this vulnerability is exploitable.

Exploit Scenario

An attacker crafts a transaction whose fine would be in excess of its balance. For example, the transaction could attempt to send an arbitrary number of messages with an insufficient amount of grams, causing the messages to fail but *without* the intended fine. This enables a denial of service attack against the TON verifiers.

Recommendations

Short term, ensure that the balance is zeroed out if there are insufficient funds to pay the gas fees and fine.

Long term, consider creating a new unsigned integer type that throws an exception on underflow and overflow and using this type to represent balances.

5. Use of deprecated OpenSSL APIs	
Severity: **Informational**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-TVMUP-5
Target: `tdutils/td/utils/{BigNum.cpp, crypto.cpp}`	

Description

TON uses several OpenSSL APIs that have been deprecated. Specifically, the `BN_is_prime_ex`, `BN_is_prime_fasttest_ex`, `AES_set_encrypt_key`, `AES_cbc_encrypt`, `SHA256_Init`, `SHA256_Update`, `SHA256_Final`, and `MD5` APIs have been deprecated.

These APIs may be removed from a future version of OpenSSL, do not prevent improper or insecure configurations, and may not receive future security updates.

Exploit Scenario

An attacker exploits a security vulnerability in one of these APIs that was not patched because the API is no longer supported.

Recommendations

Modern OpenSSL has a newer "Envelope" (EVP) interface that provides a consistent and abstracted API for various cryptographic operations, including symmetric encryption, message digests, public key encryption, and digital signatures.

The EVP API allows developers to work with different cryptographic algorithms without having to directly interact with the low-level implementation details. It provides a unified interface for cryptographic operations, making it easier to write secure and portable code.

By using the EVP interface, developers can write code that is not tied to a specific algorithm, making it more flexible and adaptable to different requirements and configurations. It abstracts the complexity of cryptographic operations and provides a higher level of security by handling various aspects, such as key management and algorithm selection, in a standardized manner.

Short term, remove all uses of deprecated functions and switch to the EVP interface.

Long term, implement static analysis in CI to detect and reject modifications that would introduce code that uses deprecated APIs.

6. MULADDDIVMOD and related instructions have unclear behavior

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TVMUP-6
Target: `crypto/vm/arithops.cpp`	

Description

The instruction `MULADDDIVMOD` is defined to have the following stack behavior:
`x y w z - q=floor((xy+w)/z) r=(xy+w)-zq`.

Consider the Fift code in figure 6.1.

```
-9759082887665682234492322154775530384411992376527688831861932665504690795898
PUSHINT
13 PUSHINT
2 PUSHINT
-6 PUSHINT
MULADDDIVMODR
```

Figure 6.1: Example Fift code that successfully computes a result even though the multiplication operation overflows

Even though `x * y` is less than -2^{256}, the program successfully computes the following:

```
q =
211446795899423115080666980020136491662259834824766591357008
54108593496724445

r= -2
```

It is not clear under what circumstances the `MULADDDIVMOD` and related new operations are and are not expected to produce an exception.

Multiply-then-divide operations are defined to compute the product using 513 bits (Fift manual, section 2.5). For the operations in the updated version, there is no information about the circumstances under which one can expect an exception. As some of the new operations contain both multiplication and addition, there are more potential overflow cases.

Exploit Scenario
A smart contract implements logic that assumes an exception is raised for out-of-range values. As a larger intermediate bit size could correctly compute larger values, an unexpected value is returned and the logic fails, causing the loss of funds.

Recommendations
Short term, implement test cases to verify that the expected boundary conditions hold for the respective operation. Document under what circumstance the out-of-range values are produced.

Long term, introduce a template for how instructions are described such that it clearly describes expectations on input and output range and other limitations.

7. Undefined behavior in CyclicBlobViewImpl	
Severity: **High**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-TVMUP-7
Target: `tddb/td/db/utils/BlobView.cpp`	

Description

The `CyclicBlobViewImpl` class's `view_impl` method, shown in figure 7.1, has undefined behavior on line 317 when the value of `data_.size()` is zero, as the use of the modulo operator with a zero operand is undefined behavior. See INT33-C for more information.

```
315    td::Result<td::Slice> view_impl(td::MutableSlice slice, td::uint64 offset)
override {
316     auto res = slice;
317     offset %= data_.size();
318     while (!slice.empty()) {
319       auto from = data_.as_slice().substr(offset).truncate(slice.size());
320       slice.copy_from(from);
321       slice.remove_prefix(from.size());
322       offset = 0;
323     }
324     return res;
325    }
```

Figure 7.1: The undefined behavior in `CyclicBlobViewImpl`
(`tddb/td/db/utils/BlobView.cpp#313–325`)

The `CyclicBlobView` implementation is currently used in the Torrent storage test cases. Code with undefined behavior cannot provide any guarantees for anything, so the test case cannot be trusted.

Exploit Scenario

Alice, a TON developer, decides to use `CyclicBlobViewImpl` for different use cases, but she is not aware that the implementation requires the data size to be greater than zero and that there is no check preventing misuse of the class. As a result, the code she is developing for TON invokes undefined behavior.

Recommendations

Short term, add a check that prevents the `view_impl` function from accepting a `data_.size()` of zero.

Long term, run static code analysis to detect vulnerabilities. We recommend running such analysis in CI to prevent issues that can be detected from building successfully.

8. Use of blst version with new-delete mismatch

Severity: **High**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-TVMUP-8
Target: `third-party/blst` (i.e., the `blst` submodule)	

Description

The version of `blst` that TON currently uses contains a `new-delete` mismatch in the line in figure 8.1. Such a mismatch is considered undefined behavior by the C++ standard.

The affected line was fixed, as shown in figure 8.2, in commit `327d30a` on July 18, 2023.

```
456    std::unique_ptr<limb_t> scratch{new limb_t[sz/sizeof(limb_t)]};
```

Figure 8.1: The line that contains the new-delete mismatch in the version of `blst` that TON currently uses (`bindings/blst.hpp#456`)

```
456    std::unique_ptr<limb_t[]> scratch{new limb_t[sz/sizeof(limb_t)]};
```

Figure 8.2: The fixed version of the line shown in figure 8.1 (`bindings/blst.hpp#456`)

The following text from the C++ standard[4] indicates that applying `delete` without `[]` to memory allocated by new with `[]` is undefined behavior:

> *In the first alternative (*delete object*), the value of the operand of `delete` may be a null pointer value, a pointer to a non-array object created by a previous* new-expression*, or a pointer to a subobject (1.8) representing a base class of such an object (Clause 10). If not, the behavior is undefined.*

Exploit Scenario

Alice builds the TON validator. In her build, the `new-delete` mismatch introduces a remote code execution vulnerability. Eve exploits the bug to remotely execute code on Alice's machine.

Recommendations

Short term, upgrade the version of `blst` that TON uses to version `327d30a` or later. Doing so will eliminate the `new-delete` mismatch and thus a source of undefined behavior.

Long term, take the following steps:

[4] Page 113

- Regularly build and run the tests with `-fsanitize=address`, as that is how this problem was found.
- Enable Dependabot for Git submodules. If Dependabot is not a feasible solution, use a GitHub workflow to automatically check for the latest release of each of TON's dependencies. Either of these approaches will alert the TON team to updates and bug fixes that could otherwise be missed.

9. Arithmetic opcodes handled inconsistently

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-TVMUP-9
Target: `crypto/fift/lib/Asm.fif`, `crypto/vm/arithops.cpp`, TVM upgrade documentation	

Description

Only four Q (i.e., “quiet”) variants of the 24 new arithmetic opcodes are registered. Additionally, some “dump” functions produce opcodes that do not match the documentation. Such behavior is likely to cause confusion.

The TVM upgrade documentation lists 24 new arithmetic opcodes. However, Q variants are registered for only four (figures 9.1 and 9.2). Moreover, there is no obvious criteria for how those four were selected.

```
471    x{B7A900} @Defop QADDDIVMOD
472    x{B7A901} @Defop QADDDIVMODR
473    x{B7A902} @Defop QADDDIVMODC
```

Figure 9.1: Registration of three new Q opcodes (`crypto/fift/lib/Asm.fif#471–473`)

```
476    x{B7A980} @Defop QADDMULDIVMOD
```

Figure 9.2: Registration of one new Q opcode (`crypto/fift/lib/Asm.fif#476`)

Additionally, for the `MULADDRSHIFTMOD` opcode and its variants, the upgrade documentation denotes the rounding mode before the word “`MOD`” (e.g., the “R” and “C” in `MULADDRSHIFTRMOD` and `MULADDRSHIFTCMOD`). However, the corresponding dump function writes the R or C at the end (figure 9.3).

```
std::string dump_mulshrmod(CellSlice&, unsigned args, int mode) {
  ...
  switch (args & 12) {
    ...
    case 0:
      os << "MULADDRSHIFTMOD";
      break;
  }
  if (round_mode) {
    os << "FRC"[round_mode];
  }
```

```
    ...
    return os.str();
}
```

Figure 9.3: The dump function that handles `MULADDRSHIFTMOD` (`crypto/vm/arithops.cpp#542–577`)

Exploit Scenario

Alice, a TON developer, writes code that uses an unregistered `Q` opcode, expecting the variant to exist. Alice's code does not compile. Alice wastes time and effort trying to understand why.

In another scenario, Alice builds a tool that outputs code using `dump_mulshrmod` from figure 9.3. The code that Alice's tool produces does not compile.

Recommendations

Short term, take the following steps:

- For each of the new 24 arithmetic opcodes, either register a `Q` variant or document why the variant is not registered.
- Write tests to verify that the output of each dump function is consistent with the documentation.

Taking these steps will eliminate two potential sources of confusion.

Long term, as new instructions are introduced into the TVM, ensure that the above standards are maintained. Doing so will reduce the likelihood of TON developers becoming confused.

10. Inconsistencies between arithmetic operations' implementation and specification

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TVMUP-10
Target: `crypto/vm/arithops.cpp`, TVM upgrade documentation	

Description

For ADDRSHIFT#MOD, MULADDRSHIFT#MOD, and their rounding variants, the TVM upgrade documentation indicates that the operation's z argument is taken literally from the instruction. However, what is actually used is z + 1. This discrepancy is likely to cause errors.

The TVM upgrade documentation states that the following is computed by z ADDRSHIFT#MOD:

```
q=floor((x+w)/2^z)

r=(x+w)-q*2^z
```

However, the implementation uses not z, but z + 1, as shown in figure 10.1. (Note that the value that is called z in the documentation is called y in figure 10.1.)

```
int exec_shrmod(VmState* st, unsigned args, int mode) {
 int y = -1;
 if (mode & 2) {
   y = (args & 0xff) + 1;
   args >>= 8;
 }
 ...
   tmp2.rshift(y, round_mode).normalize();
```

Figure 10.1: The implementation of ADDRSHIFT#MOD (`crypto/vm/arithops.cpp#333–367`)

Similarly, the documentation states that the following is computed by z MULADDRSHIFT#MOD:

```
q=floor((xy+w)/2^z)

r=(xy+w)-q*2^z
```

However, the implementation uses not z, but z + 1, as shown in figure 10.2.

```
int exec_mulshrmod(VmState* st, unsigned args, int mode) {
 int z = -1;
 if (mode & 2) {
   z = (args & 0xff) + 1;
   args >>= 8;
 }
 ...
     tmp.rshift(z, round_mode).normalize();
     ...
     tmp2.rshift(z, round_mode).normalize();
     ...
     tmp.normalize().mod_pow2(z, round_mode).normalize();
```

Figure 10.2: The implementation of `MULADDRSHIFT#MOD`
(`crypto/vm/arithops.cpp#488–530`)

Exploit Scenario

Alice, a TON developer, writes code that uses the `ADDRSHIFT#MOD` or `MULADDRSHIFT#MOD` instruction. Alice's code does not work correctly.

Recommendations

Short term, either correct the TVM upgrade documentation to match the implementation, or vice versa. The discrepancy that currently exists is likely to cause errors.

Long term, as new instructions are introduced into the TVM, write tests to verify that the implementation matches the specification. Doing so could help to expose similar errors.

11. Missing call to normalize in ADDDIVMOD implementation

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TVMUP-11
Target: `crypto/common/bigint.hpp`, `crypto/vm/arithops.cpp`	

Description

In TON, an integer can have multiple representations. `BigInt`’s `normalize` function puts an integer into a canonical form. A call to `normalize` is missing from the implementation of `ADDDIVMOD`, which can cause subsequent comparisons to produce incorrect results.

`BigInt`’s `normalize` function puts an integer into a canonical form, such as by ensuring that the digits it stores internally are within certain bounds (figure 11.1).

```
template <class Tr>
bool AnyIntView<Tr>::normalize_bool_any() {
 word_t val = 0;
 int i;
 if (!is_valid()) {
   return false;
 }
 for (i = 0; i < size() && digits[i] < Tr::Half && digits[i] >= -Tr::Half;
   i++) {
 ...
}
```

Figure 11.1: An excerpt of `normalize_bool_any`, which is called by `normalize` (`crypto/common/bigint.hpp#691–719`)

Many arithmetic operation implementations in the TVM codebase already include calls to `normalize` (figure 11.2).

```
int exec_muldivmod(VmState* st, unsigned args, int quiet) {
 ...
 switch (d) {
   case 1:
     stack.push_int_quiet(td::make_refint(quot.normalize()), quiet);
     break;
   case 3:
     stack.push_int_quiet(td::make_refint(quot.normalize()), quiet);
     // fallthrough
   case 2:
     stack.push_int_quiet(td::make_refint(tmp), quiet);
     break;
```

```
 }
 return 0;
}
```

Figure 11.2: An excerpt of `exec_muldivmod`
(`crypto/vm/arithops.cpp#425–462`)

However, the implementation of `ADDDIVMOD` lacks such a call (figure 11.3).

```
int exec_divmod(VmState* st, unsigned args, int quiet) {
 ...
 if (add) {
   ...
   stack.push_int_quiet(td::make_refint(quot), quiet);
   stack.push_int_quiet(td::make_refint(tmp), quiet);
 } else {
   ...
 }
 return 0;
}
```

Figure 11.3: In the implementation of `ADDDIVMOD`, the highlighted text should likely be `quot.normalize()`.
(`crypto/vm/arithops.cpp#266–307`)

The lack of such a call can cause problems since, for example, `BigInt`'s `cmp` function compares its operands without normalizing them (figure 11.4).

```
template <class Tr>
int AnyIntView<Tr>::cmp_any(const AnyIntView<Tr>& yp) const {
  if (yp.size() < size()) {
    return top_word() < 0 ? -1 : 1;
  } else if (yp.size() > size()) {
    return yp.top_word() > 0 ? -1 : 1;
  }
  for (int i = size() - 1; i >= 0; i--) {
    if (digits[i] < yp.digits[i]) {
      return -1;
    } else if (digits[i] > yp.digits[i]) {
      return 1;
    }
  }
  return 0;
}
```

Figure 11.4: The cmp_any function, which is called by `BigInt`'s cmp function
(`crypto/common/bigint.hpp#1079–1094`)

The issue can be reproduced with the Fift code in figure 11.5.

```
{
  =: ans-r =: ans-q
  =: z =: w  =: x
  @' x  @' w @' z
  <b x{A901} s, b> <s 0 runvmx
  .s
  abort"Exitcode != 0"
  @' ans-r <> abort"Incorrect r"
  @' ans-q <> abort"Incorrect q"
} : test

8458444444444444444444444444444444444444448888888888888847
4444444444444444444477788880 -7988888888888888888888
-10587810848400556328245438454357052079 290667056403906279575 test
```

Figure 11.5: This is the Fift code that reproduces the issue. The code errors with "Incorrect q" because, even though -10587810848400556328245438454357052079 is the correct value, the literal's internal representation differs from the computed one.

Exploit Scenario

Alice, a TON developer, writes code that performs a computation using ADDDIVMOD and compares the result to some other value. The result of the comparison is incorrect because the result of the ADDDIVMOD operation was not normalized.

Recommendations

Short term, add a call to `normalize` as suggested in the caption of figure 11.3. Doing so will help ensure comparisons involving the results of ADDDIVMOD operations are correct.

Long term, take the following steps (which are also recommended for finding TOB-TVMUP-15):

- Regularly test the code by fuzzing it. Fuzzing revealed the bug described here.
- Ensure that all arithmetic operations have a robust set of unit tests. It is possible that better unit tests could have revealed this bug.
- Use an integer type that does not allow multiple representations for an integer. The current type seems to be a common source of errors.

12. Use of deprecated cryptographic APIs	
Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-TVMUP-12
Target: `tdutils/td/utils/crypto.cpp`	

Description

The TON cryptographic API exposes unsafe and deprecated functions like MD5 (figure 12.1).

```
782    void md5(Slice input, MutableSlice output) {
783     CHECK(output.size() >= MD5_DIGEST_LENGTH);
784     auto result = MD5(input.ubegin(), input.size(), output.ubegin());
785     CHECK(result == output.ubegin());
786    }
```

Figure 12.1: TON's cryptographic utility library exposes the MD5 hash.
(`tdutils/td/utils/crypto.cpp#782–786`)

MD5 is no longer considered cryptographically secure and can be trivially collided.

This finding is of informational severity because this code does not appear to be used anywhere in TON.

Exploit Scenario

A future TON developer notices the API and decides to use a deprecated function like MD5, causing a security flaw.

Recommendations

Short term, remove the implementation of MD5 and any other deprecated cryptographic APIs from TON.

Long term, document the purpose of all cryptographic functions in TON. Regularly perform checks for unreachable code (e.g., in CI) and proactively remove dead code.

13. Bignum can segfault when converting to string or hex	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-TVMUP-13
Target: `crypto/openssl/bignum.cpp`	

Description

The `Bignum` class is used throughout TON's OpenSSL wrapper and cryptographic API. This class defines two functions to convert `Bignum` values to decimal and hexadecimal strings:

```
233    std::string Bignum::to_str() const {
234     char* ptr = BN_bn2dec(val);
235     std::string z(ptr);
236     OPENSSL_free(ptr);
237     return z;
238    }
239
240    std::string Bignum::to_hex() const {
241     char* ptr = BN_bn2hex(val);
242     std::string z(ptr);
243     OPENSSL_free(ptr);
244     return z;
245    }
```

Figure 13.1: Two functions to convert `Bignum` values to strings (`crypto/openssl/bignum.cpp#233–245`)

Note that on lines 234 and 241, the return values of `BN_bn2dec` and `BN_bn2hex` are not checked. These functions will return `nullptr` if their argument—the underlying OpenSSL big number `val`—is invalid. The pointer returned by these functions, `ptr`, is immediately passed to the `std::string` constructor on lines 235 and 242. If the value of `ptr` is `nullptr`, then the string constructor will throw a logic error and the program will segfault.

The severity of this finding is informational because the code in figure 13.1 appears to be reachable only from `test_ed25519_impl`.

Exploit Scenario

An attacker creates a malicious contract that creates an invalid `Bignum`. When this value is logged, the validators crash.

Recommendations

Short term, add a check to ensure that the return value from the `BN_` functions is not null.

Long term, consider running CodeQL over the codebase regularly. CodeQL revealed this bug. Running it regularly could reveal similar ones.

14. Risk of infinite loop during RaptorQ FEC	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Denial of Service	Finding ID: TOB-TVMUP-14
Target: `tdfec/td/fec/raptorq/Rfc.h`	

Description

TON's forward error correction (FEC) implementation has a templated convenience function for iterating over an encoding's rows:

```
61    template <class F>
62    void encoding_row_for_each(EncodingRow t, F &&f) const {
63      f(t.b);
64      for (uint16 j = 1; j < t.d; ++j) {
65        t.b = (t.b + t.a) % W;
66        f(t.b);
67      }
68
69      while (t.b1 >= P)
70        t.b1 = (t.b1 + t.a1) % P1;
71      f(W + t.b1);
72      for (uint16 j = 1; j < t.d1; ++j) {
73        t.b1 = (t.b1 + t.a1) % P1;
74        while (t.b1 >= P)
75          t.b1 = (t.b1 + t.a1) % P1;
76        f(W + t.b1);
77      }
78    }
```

Figure 14.1: A convenience function to iterate over an encoding row (`tdfec/td/fec/raptorq/Rfc.h#61–78`)

Note that on line 64, the iterator, `j`, is a `uint16`; however, the invariant `t.d` is a `uint32`. If `t.d` is greater than or equal to 2^{16} (the maximum value representable with a `uint16`), then `j` will overflow and wrap back to zero. This will result in an infinite loop.

The severity and difficulty of this finding are undetermined because it is unclear what the maximum value of `t.d` (the LT degree) can be. A code comment suggests that it is bounded above by 30, but there is no such bound explicitly specified in RFC 6330. If it is in fact provable that `t.d` will never be greater than or equal to 2^{16}, then this finding would be of informational severity.

Exploit Scenario
An attacker discovers a way to induce the LT degree to be greater than 2^{16}, causing the victim's node to enter an infinite loop.

Recommendations
Short term, change the loop iterators to be of type `uint32`.

Long term, confirm the necessity of using types narrower than 64 bits. In almost all cases, using a 64-bit type on a 64-bit architecture will be more performant than using a narrower type. The only motivation to use a narrower type would be to reduce memory usage of the structs.

15. Missing to call to normalize in MULADDRSHIFT#MOD implementation

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TVMUP-15
Target: `crypto/vm/arithops.cpp`	

Description

In TON, an integer can have multiple representations. `BigInt`'s `normalize` function puts an integer into a canonical form. A call to `normalize` is missing from the implementation of `MULADDRSHIFT#MOD`, which can cause subsequent comparisons to produce incorrect results.

`BigInt`'s `normalize` function puts an integer into a canonical form, such as by ensuring that the digits it stores internally are within certain bounds (see figure 11.1 in TOB-TVMUP-11).

However, the implementation of `MULADDRSHIFT#MOD` lacks such a call (figure 15.1).

```
488    int exec_mulshrmod(VmState* st, unsigned args, int mode) {
        ...
495     unsigned d = (args >> 2) & 3;
        ...
523     switch (d) {
524       case 1:
525         tmp.rshift(z, round_mode).normalize();
526         stack.push_int_quiet(td::make_refint(tmp), mode & 1);
527         break;
528       case 3: {
529         typename td::BigInt256::DoubleInt tmp2{tmp};
530         tmp2.rshift(z, round_mode).normalize();
531         stack.push_int_quiet(td::make_refint(tmp2), mode & 1);
532       }
533         // fallthrough
534       case 2:
535         tmp.normalize().mod_pow2(z, round_mode).normalize();
536         stack.push_int_quiet(td::make_refint(tmp), mode & 1);
537         break;
538     }
539     return 0;
540    }
```

Figure 15.1: In the implementation of `MULADDRSHIFT#MOD`, the highlighted text should likely be `tmp2.normalize()`. (`crypto/vm/arithops.cpp#488–540`)

Without the call to `normalize`, a call to `sgn` in `rshift_any` (figure 15.2) can return the wrong result.

```
template <class Tr>
bool AnyIntView<Tr>::rshift_any(int exponent, int round_mode) {
 if (exponent < 0) {
   return invalidate_bool();
 }
 if (!exponent) {
   return true;
 }
 if (exponent > size() * word_shift + word_bits - word_shift) {
   if (!round_mode) {
     *this = 0;
   } else if (round_mode < 0) {
     *this = (sgn() < 0 ? -1 : 0);
   } else {
     *this = (sgn() > 0 ? 1 : 0);
   }
   return true;
 }
 ...
}
```

Figure 15.2: The `rshift_any` function, which is called by `BigInt`'s `rshift` function (`crypto/common/bigint.hpp#1528–1593`)

The issue can be reproduced with the Fift code in figure 15.3.

```
{
  =: ans-r =: ans-q
  =: w =: y =: x
  @' x @' y @' w
  <b x{A9B0FF} s, b> <s 0 runvmx
  .s
  abort"Exitcode != 0"
  @' ans-r <> abort"Incorrect r"
  @' ans-q <> abort"Incorrect q"
} : test

5 40 -7840  -1
115792089237316195423570985008687907853269984665640564039457584007913129632296 test
```

Figure 15.3: This is the Fift code that reproduces the issue. The code errors with "Incorrect q" because it expects the highlighted `-1` to be `0`.

Exploit Scenario

Alice, a TON developer, writes code that performs a computation using `MULADDRSHIFT#MOD` and compares the result to some other value. The result of the comparison is incorrect because the result of an intermediate computation in `MULADDRSHIFT#MOD` was not normalized.

Recommendations

Short term, add a call to `normalize` as suggested in the caption of figure 15.1. Doing so will help ensure comparisons involving the results of `MULADDRSHIFT#MOD` operations are correct.

Long term, take the following steps (which are also recommended for finding TOB-TVMUP-11):

- Regularly test the code by fuzzing it. Fuzzing revealed the bug described here.
- Ensure that all arithmetic operations have a robust set of unit tests. It is possible that better unit tests could have revealed this bug.
- Use an integer type that does not allow multiple representations for an integer. The current type seems to be a common source of errors.

16. BLS gas costs are inconsistent with specification

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TVMUP-16
Target: `crypto/vm/tonops.cpp`, TVM upgrade documentation	

Description

The gas costs for various BLS operations do not match the costs indicated in the specification. Improperly implemented gas costs could be used for griefing attacks. Such discrepancies can also cause confusion for users.

As an example, the gas cost for `BLS_G1_ADD` is listed in the documentation as 3,959. However, the gas cost that is actually applied for this operation is 3,934. The `bls_g1_add_sub_gas_price` function adds 3,900 (figures 16.1 and 16.2) and the additional 34 is from 10 per instruction plus 24 opcode bits times 1 per opcode bit (figure 16.3 and 16.4).

```
int exec_bls_g1_add(VmState* st) {
 VM_LOG(st) << "execute BLS_G1_ADD";
 Stack& stack = st->get_stack();
 stack.check_underflow(2);
 st->consume_gas(VmState::bls_g1_add_sub_gas_price);
 bls::P1 b = slice_to_bls_p1(*stack.pop_cellslice());
 bls::P1 a = slice_to_bls_p1(*stack.pop_cellslice());
 stack.push_cellslice(bls_to_slice(bls::g1_add(a, b).as_slice()));
 return 0;
}
```

Figure 16.1: The implementation of BLS_G1_ADD
(`crypto/vm/tonops.cpp#893–902`)

```
bls_g1_add_sub_gas_price = 3900,
```

Figure 16.2: The gas cost implemented for BLS_G1_ADD
(`crypto/vm/vm.h#140`)

```
int OpcodeInstrSimple::dispatch(VmState* st, CellSlice& cs, unsigned opcode, unsigned bits) const {
 st->consume_gas(gas_per_instr + opc_bits * gas_per_bit);
 if (bits < opc_bits) {
   throw VmError{Excno::inv_opcode, "invalid or too short opcode", opcode + (bits << max_opcode_bits)};
 }
```

```
    cs.advance(opc_bits);
    return exec_instr(st, cs, opcode >> (max_opcode_bits - opc_bits),
opc_bits);
  }
```

Figure 16.3: The gas consumption for a "simple" instruction
(`crypto/vm/opctable.cpp#172–179`)

```
static constexpr unsigned gas_per_instr = 10, gas_per_bit = 1;
```

Figure 16.4: The `gas_per_instr` and `gas_per_bit` values
(`crypto/vm/opctable.h#43`)

As mentioned under Coverage Limitations, we were not able to exhaustively check all instructions' gas costs.

Exploit Scenario

Alice, a TON developer, writes code that uses the `BLS_G1_ADD` instruction. Alice's code does not consume the amount of gas she expects. Alice wastes time and effort trying to understand why.

Recommendations

Short term, ensure that the BLS operations' implemented gas costs match those indicated in their specifications. Doing so will help prevent griefing attacks and will reduce the likelihood of users becoming confused.

Long term, add tests to verify that the TVM operations' implemented gas costs match those indicated in their specifications. Doing so will help prevent such problems from arising again.

17. Use of libsodium might stall the process	
Severity: **Low**	Difficulty: **High**
Type: Timing	Finding ID: TOB-TVMUP-17
Target: `crypto/vm/tonops.cpp`	

Description

The documentation for `libsodium`'s `sodium_init` function includes a section describing how initialization of the library might stall on Linux. The `sodium_init` function is invoked during the execution of the Ristretto255 instructions in the TVM.

Exploit Scenario

`libsodium` stalls during initialization, preventing a validator node from completing before the timeout. The validator is penalized for failing to complete on time.

Recommendations

Short term, modify the relevant code so that `sodium_init` is invoked when the TVM boots to ensure that initialization of `libsodium` is complete before it is used in a time-sensitive setting.

Long term, ensure that prerequisites for and behavior of introduced dependencies are carefully scrutinized before being implemented. Furthermore, ensure that there is a process for monitoring dependencies over time to prevent unexpected failures as dependencies change.

18. RIST255_MUL uses nonstandard method for handling errors

Severity: **High**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-TVMUP-18
Target: `crypto/vm/tonops.cpp`	

Description

The documentation for the `crypto_scalarmult_ristretto255` function states that zero is the only return value that indicates success. TON's invocation of the function can be found in the `exec_ristretto255_mul` function and the relevant excerpt is shown in figure 18.1. When `crypto_scalarmult_ristretto255` fails to compute the scalar multiplication and returns with an error, the code checks whether the output buffer `rb` has been modified and, if so, continues to the success branch instead of returning an error.

```
722    unsigned char xb[32], nb[32], rb[32];
723    memset(rb, 255, sizeof(rb));
724    CHECK(sodium_init() >= 0);
725    if (!x->export_bytes(xb, 32, false) || !export_bytes_little(n, nb) ||
crypto_scalarmult_ristretto255(rb, nb, xb)) {
726      if (std::all_of(rb, rb + 32, [](unsigned char c) { return c == 255; })) {
727        if (quiet) {
728          stack.push_bool(false);
729          return 0;
730        }
731        throw VmError{Excno::range_chk, "invalid x or n"};
732      }
733    }
734    td::RefInt256 r{true};
```

Figure 18.1: An excerpt of `exec_ristretto255_mul` showing unexpected error handling behavior (`crypto/vm/tonops.cpp#722–734`)

The difficulty of this finding is undetermined because it is unclear under what conditions the Ristretto implementation will modify the buffer.

Exploit Scenario

The implementation of `crypto_scalarmult_ristretto255` is altered in a way that causes it to modify the output buffer before returning an error code. This causes the `if` statement on line 726 to fail and bypass the error exit, thereby leading to a seemingly successful output. Because not all validators necessarily have the same version of the `libsodium` dependency, there is a risk that some will fail and some will succeed, potentially causing a fork of the chain.

Recommendations

Short term, remove the additional check to prevent failed multiplication operations from indicating success.

Long term, require unit tests, having both positive and negative tests, to accompany every feature that is introduced.

19. Cell slices for public keys and signatures can have excess data	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TVMUP-19
Target: `crypto/vm/tonops.cpp`	

Description

Figure 19.1 shows the function used to convert a generic `CellSlice` into the P1 type, which represents public keys in the TON BLS12-381 implementation. The P1 type is a `BitArray` of 384 bits. The `prefetch_bytes` function has a check to ensure that the `CellSlice` holds at least the length of the destination slice (p1 in figure 19.1). However, there is no check to ensure that the `CellSlice` is exactly 384 bits. This can lead to incompatibility issues between the TON blockchain implementation and other external entities. The same issue is present in the code that converts cell slices to the P2, FP, and FP2 types—all part of the BLS12-381 integration.

```
776    static bls::P1 slice_to_bls_p1(const CellSlice& cs) {
777     bls::P1 p1;
778     if (!cs.prefetch_bytes(p1.as_slice())) {
779       throw VmError{Excno::cell_und, PSTRING() << "slice must contain at least
" << bls::P1_SIZE << " bytes"};
780     }
781     return p1;
782    }
```

Figure 19.1: The function that converts a `CellSlice` to P1
(`crypto/vm/tonops.cpp#776–782`)

Exploit Scenario

`CellSlices` used for successful BLS12-381 operations on the TON blockchain are exported and used on a separate blockchain with stricter validation, causing the operations to fail and causing incompatibility issues.

Recommendations

Short term, implement a check in the `slice_to_bls_{p1,p2,fp,fp2}` functions that ensures the source `CellSlice` is of the exact expected length.

Long term, ensure that bounds checks are as tight as possible and include fuzz testing to ensure resiliency against data padding attacks.

20. Divergent behavior among BLS instructions when n is 0	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TVMUP-20
Target: `crypto/vm/tonops.cpp`	

Description

The instructions `BLS_G1_MULTIEXP`, `BLS_G2_MULTIEXP`, `BLS_AGGREGATE`, `BLS_FASTAGGREGATEVERIFY`, `BLS_AGGREGATEVERIFY`, and `BLS_PAIRING` all operate on multiple signatures, keys, or pairs of them. The stack value `n` is used to represent the number of values to process.

For the special case in which `n` is `0`, these functions have divergent behavior:

- `BLS_AGGREGATE` throws an exception.
- `BLS_FASTAGGREGATEVERIFY`, `BLS_AGGREGATEVERIFY`, and `BLS_PAIRING` return `False`.
- `BLS_G1_MULTIEXP` and `BLS_G2_MULTIEXP` return the zero point.

Exploit Scenario

Alice, a TON developer who has previously used `BLS_AGGREGATE`, incorrectly assumes an exception is thrown when n is `0` when calling `BLS_G1_MULTIEXP`, but instead, the zero point is returned. As a result, the smart contract code continues executing even though it should not, and funds are lost.

Recommendations

Short term, either modify these instructions so that they behave the same way when `n` is `0` or clarify the documentation concerning each instruction's expected behavior in this case.

Long term, use fuzz testing to explore system behavior under various scenarios. Furthermore, when introducing new instructions, consider studying already implemented, related instructions to align behavior with existing code.

21. Uninitialized data read when downcast_call fails	
Severity: **High**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-TVMUP-21
Target: `fec/fec.cpp`	

Description

The code in figure 21.1 uses the `downcast_call` and `overloaded` APIs. The lambda that is invoked will assign values to the previously uninitialized variables `data_size_int`, `symbol_size_int`, and `symbols_count_int`. The implementation of `downcast_call` for the `fec_Type` is shown in figure 21.2. From figure 21.2, it is evident that the default branch will not invoke the function that assigns the `data_size_int`, `symbol_size_int`, and `symbols_count_int` values. Should this branch be taken, line 108 would cause a read of the uninitialized value `data_size_int`, which is undefined behavior. Note that the variables used on lines 109 and 110 are also uninitialized.

```
101    td::Result<FecType> FecType::create(tl_object_ptr<ton_api::fec_Type> obj) {
102     td::int32 data_size_int, symbol_size_int, symbols_count_int;
103     ton_api::downcast_call(*obj, td::overloaded([&](const auto &obj) {
104       data_size_int = obj.data_size_;
105       symbol_size_int = obj.symbol_size_;
106       symbols_count_int = obj.symbols_count_;
107     }));
108     TRY_RESULT(data_size, td::narrow_cast_safe<size_t>(data_size_int));
109     TRY_RESULT(symbol_size, td::narrow_cast_safe<size_t>(symbol_size_int));
110     TRY_RESULT(symbols_count, td::narrow_cast_safe<size_t>(symbols_count_int));
```

Figure 21.1: Using `downcast_call` without checking whether the function call happened can cause a read of uninitialized values. (`fec/fec.cpp#101–110`)

```
/**
 * Calls specified function object with the specified object downcasted to the
most-derived type.
 * \param[in] obj Object to pass as an argument to the function object.
 * \param[in] func Function object to which the object will be passed.
 * \returns whether function object call has happened. Should always return
true for correct parameters.
 */
template <class T>
bool downcast_call(fec_Type &obj, const T &func) {
 switch (obj.get_id()) {
   case fec_raptorQ::ID:
     func(static_cast<fec_raptorQ &>(obj));
     return true;
```

```
      case fec_roundRobin::ID:
        func(static_cast<fec_roundRobin &>(obj));
        return true;
      case fec_online::ID:
        func(static_cast<fec_online &>(obj));
        return true;
      default:
        return false;
    }
  }
```

Figure 21.2: The `downcast_call` implementation for type `fec_Type` (`tl/generate/auto/tl/ton_api.hpp`)

Exploit Scenario

An adversarial TON user identifies a scenario in which `obj` can be made to hold a value with an `ID` that is not recognized by `downcast_call`. This causes `data_size_int` to be uninitialized and later read, invoking undefined behavior and leading to a crash or other severe outcome.

Recommendations

Short term, implement a check of the return value of `downcast_call` that will propagate a failed call to the `td::Result<FecType>` to indicate failure.

Long term, mark the return value of `downcast_call` as a `nodiscard` to indicate to developers that they should account for cases in which `func` is not invoked. Furthermore, run static code analysis in CI to prevent developers from being able to introduce uninitialized reads that can be automatically detected.

22. Register c7 tuple element "previous blocks" can be null	
Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-TVMUP-22
Target: `crypto/block/transaction.cpp`, documentation	

Description

The code in figure 22.1 is used to initialize the tuple residing in register c7. The documentation states that this tuple includes an element with information about previous blocks.

However, based on the implementation, this tuple element might be an empty stack entry.

```
 954    tuple.push_back(cfg.prev_blocks_info.not_null() ?
vm::StackEntry(cfg.prev_blocks_info) : vm::StackEntry());
```

Figure 22.1: The initialization of the c7 register, with tuple index 13 holding information about previous blocks (`crypto/block/transaction.cpp#954`)

Exploit Scenario

Alice, a TON developer, relies on information about previous blocks but is not aware that the code might return an empty stack entry, causing unexpected behavior in her code, potentially leading to the loss of funds.

Recommendations

Short term, document that the information about previous blocks is not necessarily available and what to expect when it is not.

Long term, implement test cases and ensure that they match the documentation to prevent users from getting unexpected values that are not stated in the documentation.

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
Front-Running Resistance	The system's resistance to front-running attacks
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

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Data Used for TOB-TVMUP-2

This appendix contains the raw data used to calculate the percentage of lines in the `crypto` subdirectory's `.cpp` files that have line comments, as described in finding TOB-TVMUP-2. Specifically, table C.1 contains the number of lines matching the regular expression `//.*`, as well as the total number of lines, for each file ending in `.cpp` in the `crypto` subdirectory.

Source File	Lines Matching //.*	Total Lines	Percentage
`Ed25519.cpp`	5	402	1.24%
`block/Binlog.cpp`	11	493	2.23%
`block/adjust-block.cpp`	1	208	0.48%
`block/block-auto.cpp`	638	25,812	2.47%
`block/block-db.cpp`	25	843	2.97%
`block/block-parse.cpp`	563	2,328	24.18%
`block/block.cpp`	109	2,300	4.74%
`block/check-proof.cpp`	18	666	2.70%
`block/create-state.cpp`	86	949	9.06%
`block/dump-block.cpp`	4	351	1.14%
`block/mc-config.cpp`	97	2,261	4.29%
`block/output-queue-merger.cpp`	3	221	1.36%
`block/test-block.cpp`	2	248	0.81%

block/test-weight-distr.cpp	12	199	6.03%
block/transaction.cpp	389	2,809	13.85%
common/bigexp.cpp	10	261	3.83%
common/bigint.cpp	3	41	7.32%
common/bitstring.cpp	20	668	2.99%
common/refcnt.cpp	3	60	5.00%
common/refint.cpp	11	379	2.90%
common/util.cpp	2	236	0.85%
ellcurve/Ed25519.cpp	11	280	3.93%
ellcurve/Fp25519.cpp	2	33	6.06%
ellcurve/Montgomery.cpp	5	138	3.62%
ellcurve/TwEdwards.cpp	12	255	4.71%
ellcurve/p256.cpp	2	91	2.20%
ellcurve/secp256k1.cpp	1	42	2.38%
fift/Continuation.cpp	55	535	10.28%
fift/Dictionary.cpp	8	128	6.25%
fift/Fift.cpp	2	82	2.44%
fift/HashMap.cpp	2	371	0.54%

fift/IntCtx.cpp	3	306	0.98%
fift/SourceLookup.cpp	2	89	2.25%
fift/fift-main.cpp	3	227	1.32%
fift/utils.cpp	3	223	1.35%
fift/words.cpp	98	3,545	2.76%
func/abscode.cpp	6	528	1.14%
func/analyzer.cpp	41	916	4.48%
func/asmops.cpp	2	375	0.53%
func/builtins.cpp	42	1,265	3.32%
func/codegen.cpp	21	912	2.30%
func/func-main.cpp	1	129	0.78%
func/func.cpp	4	261	1.53%
func/gen-abscode.cpp	7	451	1.55%
func/keywords.cpp	4	137	2.92%
func/optimize.cpp	9	654	1.38%
func/parse-func.cpp	36	1,818	1.98%
func/stack-transform.cpp	77	1,056	7.29%
func/unify-types.cpp	6	431	1.39%

funcfiftlib/funcfiftlib.cpp	13	173	7.51%
keccak/keccak.cpp	2	473	0.42%
openssl/bignum.cpp	4	261	1.53%
openssl/rand.cpp	7	122	5.74%
openssl/residue.cpp	5	176	2.84%
parser/lexer.cpp	4	338	1.18%
parser/srcread.cpp	3	230	1.30%
parser/symtable.cpp	5	181	2.76%
smartcont/auto/config-code.cpp	1	1	100.00%
smartcont/auto/dns-auto-code.cpp	0	1	0.00%
smartcont/auto/dns-manual-code.cpp	1	1	100.00%
smartcont/auto/elector-code.cpp	1	1	100.00%
smartcont/auto/highload-wallet-code.cpp	1	1	100.00%
smartcont/auto/highload-wallet-v2-code.cpp	1	1	100.00%
smartcont/auto/multisig-code.cpp	1	1	100.00%
smartcont/auto/payment-channel-code.cpp	0	1	0.00%
smartcont/auto/pow-testgiver-code.cpp	1	1	100.00%

smartcont/auto/restricted-wallet-code.cpp	1	1	100.00%
smartcont/auto/restricted-wallet2-code.cpp	0	1	0.00%
smartcont/auto/restricted-wallet3-code.cpp	0	1	0.00%
smartcont/auto/simple-wallet-code.cpp	1	1	100.00%
smartcont/auto/simple-wallet-ext-code.cpp	0	1	0.00%
smartcont/auto/wallet-code.cpp	1	1	100.00%
smartcont/auto/wallet3-code.cpp	0	1	0.00%
smc-envelope/GenericAccount.cpp	4	163	2.45%
smc-envelope/HighloadWallet.cpp	2	90	2.22%
smc-envelope/HighloadWalletV2.cpp	2	107	1.87%
smc-envelope/ManualDns.cpp	38	634	5.99%
smc-envelope/MultisigWallet.cpp	2	198	1.01%
smc-envelope/PaymentChannel.cpp	4	291	1.37%
smc-envelope/SmartContract.cpp	37	351	10.54%
smc-envelope/SmartContractCode.cpp	16	192	8.33%
smc-envelope/TestGiver.cpp	5	66	7.58%

smc-envelope/TestWallet.cpp	3	106	2.83%
smc-envelope/Wallet.cpp	4	110	3.64%
smc-envelope/WalletInterface.cpp	2	80	2.50%
smc-envelope/WalletV3.cpp	2	86	2.33%
test/Ed25519.cpp	17	219	7.76%
test/fift.cpp	1	165	0.61%
test/modbigint.cpp	12	1,074	1.12%
test/test-bigint.cpp	95	876	10.84%
test/test-cells.cpp	12	656	1.83%
test/test-db.cpp	68	2,096	3.24%
test/test-ed25519-crypto.cpp	22	314	7.01%
test/test-smartcont.cpp	85	1,661	5.12%
test/vm.cpp	22	450	4.89%
tl/tlbc-gen-cpp.cpp	103	3,465	2.97%
tl/tlbc.cpp	85	3,167	2.68%
tl/tlblib.cpp	3	387	0.78%
util/Miner.cpp	3	129	2.33%
util/pow-miner.cpp	9	245	3.67%

vm/Hasher.cpp	1	148	0.68%
vm/arithops.cpp	8	1,052	0.76%
vm/atom.cpp	2	97	2.06%
vm/bls.cpp	5	334	1.50%
vm/boc.cpp	65	1,217	5.34%
vm/cellops.cpp	3	1,458	0.21%
vm/cells/Cell.cpp	2	59	3.39%
vm/cells/CellBuilder.cpp	2	628	0.32%
vm/cells/CellHash.cpp	2	28	7.14%
vm/cells/CellSlice.cpp	15	1,132	1.33%
vm/cells/CellString.cpp	2	212	0.94%
vm/cells/CellTraits.cpp	2	45	4.44%
vm/cells/CellUsageTree.cpp	8	136	5.88%
vm/cells/DataCell.cpp	14	368	3.80%
vm/cells/LevelMask.cpp	3	32	9.38%
vm/cells/MerkleProof.cpp	6	421	1.43%
vm/cells/MerkleUpdate.cpp	41	513	7.99%
vm/continuation.cpp	52	662	7.85%

`vm/contops.cpp`	17	1,227	1.39%
`vm/cp0.cpp`	11	51	21.57%
`vm/db/BlobView.cpp`	6	181	3.31%
`vm/db/CellStorage.cpp`	5	163	3.07%
`vm/db/DynamicBagOfCellsDb.cpp`	20	564	3.55%
`vm/db/StaticBagOfCellsDb.cpp`	20	544	3.68%
`vm/db/TonDb.cpp`	22	325	6.77%
`vm/debugops.cpp`	5	162	3.09%
`vm/dict.cpp`	284	2,954	9.61%
`vm/dictops.cpp`	3	822	0.36%
`vm/dispatch.cpp`	3	71	4.23%
`vm/large-boc-serializer.cpp`	14	412	3.40%
`vm/memo.cpp`	2	33	6.06%
`vm/opctable.cpp`	4	470	0.85%–
`vm/stack.cpp`	46	1,004	4.58%
`vm/stackops.cpp`	5	589	0.85%
`vm/tonops.cpp`	80	1,847	4.33%
`vm/tupleops.cpp`	2	402	0.50%

`vm/utils.cpp`	3	152	1.97%
`vm/vm.cpp`	42	773	5.43%
Total	**4,015**	**97,649**	**4.11%**

Table C.1: The raw data used to perform the calculations referenced in finding TOB-TVMUP-2

D. Non-Security-Related Findings

The following recommendations are not associated with specific vulnerabilities. However, implementing them may enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Dead initialization in `lexer.cpp`**
  The `begin` variable is never used after it is assigned. Consider removing the assignment or the entire statement if there are no side effects in `src.get_ptr()`.

```
251    if (is_multiline_quote(src.get_ptr(), src.get_end_ptr())) {
252     src.advance(multiline_quote.size());
253     const char* begin = src.get_ptr();
254     const char* end = nullptr;
255     SrcLocation here = src.here();
256     std::string body;
257     while (!src.is_eof()) {
```

Figure D.1: The dead initialization on line 253 (`crypto/parser/lexer.cpp#251–257`)

- **Erroneous assignment of `status` to `OK` (or unclear code)**
  In the code in figure D.2, if the true branch is taken, `status` is "moved from." In this particular case, the move will cause `status` to be reset to `OK`. Following the `if` body, `status` is returned, causing the function to return `OK` even though `status` was an error. Either the function returns an incorrect status, in which case this is a bug, or the code could be improved to clarify that this is the intended behavior. Consider using explicit function names that clarify the code's behavior for cases in which values are used without being reset in any way after `move` operations.

```
107    if (status.is_error() && !UdpSocketFd::is_critical_read_error(status)) {
108     queue.push(UdpMessage{{}, {}, std::move(status)});
109    }
   110    return status;
```

Figure D.2: The returning of `status`, which is used after the `move` operation (`tdutils/td/utils/BufferedUdp.h#107–110`)

- **Incorrect error message due to use of JSON value after `move`**
  In the code in figure D.3, the variable `from` is "moved from" on line 111. If the result of `status.is_ok()` on line 112 is not true, `from` will be used after the move. The value of `from` will be reset to the `Type::Null` type, resulting in an incorrect error message. Consider having the type extracted before the move from the variable `from`.

```
111    auto status = from_json(x, std::move(from));
```

```
  if (status.is_ok()) {
    to = x != 0;
    return Status::OK();
  }
    return Status::Error(PSLICE() << "Expected bool, got " << from.type());
```

Figure D.3: The `from` variable is used after the `move` operation, causing an incorrect error printout. (`tl/tl/tl_json.h#111–116`)

- **Redundant stores to `var0` in generated code**
  The following is one example of generated code in `tl/generate/auto/tl/lite_api.cpp`. The `store` methods contain a redundant `var0` variable, which is stored to but not used outside of the `store` expression. Consider removing the generation of the code that stores to `var0` and, in this example, use `mode_` directly.

```
void liteServer_validatorStats::store(td::TlStorerUnsafe &s) const {
(void)sizeof(s);
std::int32_t var0;
TlStoreBinary::store((var0 = mode_), s);
TlStoreObject::store(id_, s);
TlStoreBinary::store(count_, s);
TlStoreBool::store(complete_, s);
TlStoreString::store(state_proof_, s);
TlStoreString::store(data_proof_, s);
}
```

Figure D.4: The redundant store to `var0`, which is never read outside of the expression (`tl/generate/auto/tl/lite_api.cpp`)

- **Dead assignments**
  The following are assignments that have no impact on the behavior of the program (i.e., they are dead). Consider removing the assignments, as they have no impact on the program behavior. Furthermore, consider running static code analysis to automatically detect dead assignments and keep the codebase tidy.
  - The assignment to op on line 2255 in `crypto/tl/tlbc.cpp`
  - The assignment to op on line 2259 in `crypto/tl/tlbc.cpp`
  - The assignment to `alive_end` on line 403 in `storage/test/storage.cpp`
  - The assignment to ok on line 521 in `storage/test/storage.cpp`
  - The assignment to `file_size` on line 223 in `tddb/td/db/utils/BlobView.cpp`

    - The assignment to `prev_skip` on line 162 in `crypto/tl/tlbc-gen-cpp.cpp`

- **Use of non-nullable pointer rather than reference**
  The code excerpt in figure D.5 includes a parameter named `st`, which is a pointer type. However, the code does not check to ensure that `st` is not null. Either the check is missing and this code is susceptible to undefined behavior, or the author knows that `st` cannot be null. If the latter case is correct, consider making the `st` variable a reference to convey that fact to the reader.

```
609    int exec_ristretto255_from_hash(VmState* st) {
610     VM_LOG(st) << "execute RIST255_FROMHASH";
611     Stack& stack = st->get_stack();
612     stack.check_underflow(2);
```

Figure D.5: The `st` variable is assumed not to be null and could have been a reference. (`crypto/vm/tonops.cpp#609–612`)

- **Superfluous `std::max` call in `exec_bls_aggregate`**
  In the code in figure D.6, the call to `pop_smallint_range` on line 847 guarantees that n is greater than or equal to 1. Furthermore, the gas prices are positive, so the `std::max` call is not needed because the cost will always be greater than zero. Consider simplifying the expression to improve the code's readability.

```
846    Stack& stack = st->get_stack();
847    int n = stack.pop_smallint_range(stack.depth() - 1, 1);
848    st->consume_gas(
849       std::max(0LL, VmState::bls_aggregate_base_gas_price + (long long)n *
VmState::bls_aggregate_element_gas_price));
850    std::vector<bls::P2> sigs(n);
```

Figure D.6: A superfluous `std::max` call when arguments are always greater than zero (`crypto/vm/tonops.cpp#846–850`)

- **Allocation of 64 bytes for a curve point occupying 32 bytes in `exec_ristretto255_validate`**
  The code in figure D.7 converts a TVM integer into a curve point for use with `ristretto255`. The point representation for `ristretto255` in `libsodium` requires a 32-byte buffer. The `exec_ristretto255_validate` function allocates 64 bytes for the curve point. Although this will function correctly, we recommend changing the size to 32 bytes to prevent future issues.

```
634    auto x = stack.pop_int();
635    st->consume_gas(VmState::rist255_validate_gas_price);
636    unsigned char xb[64];
637    CHECK(sodium_init() >= 0);
```

```
638    if (!x->export_bytes(xb, 32, false) ||
!crypto_core_ristretto255_is_valid_point(xb)) {
```

Figure D.7: The excess allocation for a curve point(`crypto/vm/tonops.cpp#634–638`)

- **Reference to nonexistent instruction SETLIBRARY in documentation**
  The TVM upgrade documentation refers to an instruction called SETLIBRARY that has been altered to accept additional values for mode. However, there is no instruction called SETLIBRARY implemented in the codebase; the SETLIBCODE and CHANGELIB operations are likely the intended subjects of this documentation. The upgrade documentation further states that SETLIBRARY will not work if +2 is used. We did not find any evidence of that. The values we deem acceptable for mode when used in these instructions are 0, 1, 2, 16, 17 and 18. Consider updating the documentation or code to prevent any confusion.

- **Missing n stack argument for several BLS instructions**
  The documentation for BLS_FASTAGGREGATEVERIFY, BLS_G1_MULTIEXP, and BLS_G2_MULTIEXP is missing the n stack argument indicating the number of keys, signatures, or pairs of keys and signatures to process.

- **Use of C-style casts**
  C-style casts are used in places where C++-style casts (e.g., `static_cast`, `reinterpret_cast`, and `const_cast`) could be used. An example appears in figure D.8. C++-style casts have the advantage that they are restricted in terms of the types they can convert, and those restrictions are enforced at compile time.

```
124    void bits_memcpy(unsigned char* to, int to_offs, const unsigned char* from,
int from_offs, std::size_t bit_count) {
        ...
133     int sz = (int)bit_count;
```

Figure D.8: An example of code using a C-style cast where a C++-style cast could be used (`crypto/common/bitstring.cpp#124–133`)

- **Unused argument**
  The `preload_fift` argument in figure D.9 is unused. The argument can be removed if it is not necessary.

```
36    td::Status run_fift(std::string name, bool expect_error = false, bool
preload_fift = true) {
37      auto res = fift::mem_run_fift(load_test(name));
38      if (expect_error) {
39        res.ensure_error();
40        return td::Status::OK();
41      }
42      res.ensure();
43      REGRESSION_VERIFY(res.ok().output);
```

```
    return td::Status::OK();
  }
```

Figure D.9: The `run_fift` function, which has the unused `preload_fift` argument (`crypto/test/fift.cpp#36–45`)

- **Unnamed constants**
  Several functions in `bigint.hpp` take a `round_mode` argument that is expected to hold -1 (floor), 0 (round), or 1 (ceiling). An example appears in figure D.10. The code would be more clear if the `round_mode` arguments were an `enum` type rather than an `int`.

```
template <class Tr>
bool AnyIntView<Tr>::mod_div_any(const AnyIntView<Tr>& yp, AnyIntView<Tr>& quot, int round_mode) {
    ...
      if (!round_mode) {
        if ((yv > 0 && rem * 2 >= yv) || (yv < 0 && rem * 2 <= yv)) {
          rem -= yv;
          digits[0]++;
        }
      } else if (round_mode > 0 && rem) {
        rem -= yv;
        digits[0]++;
      }
```

Figure D.10: An example function in `bigint.hpp` with a `round_mode` argument (`crypto/common/bigint.hpp#1278–1298`)

- **Inconsistently assigned `round_mode` variables**
  As explained in the previous bullet point, several functions in `bigint.hpp` take a `round_mode` argument that is expected to be -1, 0, or 1. In many cases, those arguments are fed from `round_mode` variables assigned in `arithops.cpp`. However, `arithops.cpp` assigns to the `round_mode` variables inconsistently (e.g., figures D.11 and D.12). Inconsistent assignments to `round_mode` variables increase the risk of confusion.

```
  int round_mode = (int)(args & 3) - 1;
```

Figure D.11: One example assignment to a `round_mode` variable in `arithops.cpp` (`crypto/vm/arithops.cpp#267`)

```
  int round_mode = (int)(args & 3);
```

Figure D.12: Another example assignment to a `round_mode` variable in `arithops.cpp` (`crypto/vm/arithops.cpp#310`)

- **Typos in TVM upgrade documentation**
  The TVM upgrade documentation contains the following two typographical errors:

  - In the descriptions of `BLS_G1_MULTIEXP` and `BLS_G2_MULTIEXP`, the text "and scalars `n_i`" should be "and scalars `s_i`".

  - In the description of `BLS_AGGREGATEVERIFY`, the text "kay-message pairs" should be "key-message pairs".

E. Keccak Fuzzing Code

Figure E.1 contains the code used to fuzz the TVM's Keccak-512 implementation. A slight modification was used to fuzz the Keccak-256 implementation.

The code in figure E.1 works roughly as follows. It reads a set of sequences from standard input. Each sequence is expected to consist of the following:

- A single byte, meant to represent a number of bits
- `ceiling(length / 8)` many bytes, where `length` is the number of bits (the byte mentioned in the point above)

The sequences are hashed using the TVM's Keccak implementation. The sequences are similarly hashed using a slight variant of the SHA-3 reference implementation (see below). Finally, the results returned by the two implementations are compared. If the results differ, the program aborts.

Note that changes are needed to convert a SHA-3 implementation to Keccak. The changes required for the SHA-3 reference implementation appear in figure E.2.

AFLplusplus was the fuzzing engine used to run the code in figure E.1. The code was run with a trivial (essentially meaningless) initial corpus.

```
#include <err.h>
#include <unistd.h>

#include "vm/Hasher.h"
#include "vm/excno.hpp"

extern "C" {
#include "third-party/XKCP/bin/reference/libXKCP.a.headers/KeccakHash.h"
}

using namespace vm;

#define div_up(x, y) (((x) + (y)-1) / (y))

const size_t BUF_SIZE = 8192;
// const size_t HASH_SIZE = 32;
const size_t HASH_SIZE = 64;

const uint8_t *ton_keccak(const uint8_t *buf, size_t size);
const uint8_t *xkcp_keccak(const uint8_t *buf, size_t size);
void dump(const char *label, const uint8_t *buf, size_t size);

int main() {
```

```
  try {
    uint8_t buf[BUF_SIZE];
    ssize_t size = read(STDIN_FILENO, buf, sizeof(buf));
    if (size < 0) {
      err(EXIT_FAILURE, "read");
    }

    const uint8_t *ton_hash = ton_keccak(buf, size);
    const uint8_t *xkcp_hash = xkcp_keccak(buf, size);

    dump(" ton", ton_hash, HASH_SIZE);
    dump("xkcp", xkcp_hash, HASH_SIZE);

    assert(memcmp(ton_hash, xkcp_hash, HASH_SIZE) == 0);
  } catch (VmError &err) {
    printf("%s\n", err.get_msg());
    throw;
  }

  return 0;
}

template <typename F>
void consume_with(const uint8_t *buf, size_t size, F f) {
  const uint8_t *const end = buf + size;
  const uint8_t *p = buf;
  size_t bits_consumed = 0;
  while (p + 1 <= end && *p != 0 && p + 1 + div_up(*p, 8) <= end) {
    const size_t n_bytes = div_up(*p, 8);
    dump("p", p, 1 + n_bytes);
    // Uncomment the next check to restrict fuzzing to whole bytes only.
    /* if (*p % 8 != 0) {
      exit(0);
    } */
    f(p + 1, *p);
    bits_consumed += *p;
    p += 1 + n_bytes;
  }
  if (bits_consumed % 8 != 0) {
    const uint8_t x = 0;
    f(&x, 8 - (bits_consumed % 8));
  }
}

const uint8_t *ton_keccak(const uint8_t *buf, size_t size) {
  // Hasher hasher(Hasher::KECCAK256);
  Hasher hasher(Hasher::KECCAK512);

  consume_with(buf, size, [&hasher](const uint8_t *x, size_t y) {
hasher.append(x, y); });

  static td::BufferSlice hash = hasher.finish();
  return reinterpret_cast<const uint8_t *>(hash.data());
```

```
}

const uint8_t *xkcp_keccak(const uint8_t *buf, size_t size) {
 Keccak_HashInstance hash_instance;
 // Keccak_HashInitialize_SHA3_256(&hash_instance);
 Keccak_HashInitialize_SHA3_512(&hash_instance);

 consume_with(buf, size, [&hash_instance](const uint8_t *x, size_t y) { Keccak_HashUpdate(&hash_instance, x, y); });

 static BitSequence bit_sequence[HASH_SIZE];
 Keccak_HashFinal(&hash_instance, bit_sequence);

 return bit_sequence;
}

void dump(const char *label, const uint8_t *buf, size_t size) {
 printf("%s: ", label);
 for (size_t i = 0; i < size; i++) {
   printf("%02x", buf[i]);
 }
 printf("\n");
}
```

Figure E.1: The code used to fuzz the Keccak implementations

```
diff --git a/lib/high/Keccak/FIPS202/KeccakHash.h b/lib/high/Keccak/FIPS202/KeccakHash.h
index e99d99d..5768049 100644
--- a/lib/high/Keccak/FIPS202/KeccakHash.h
+++ b/lib/high/Keccak/FIPS202/KeccakHash.h
@@ -70,7 +70,7 @@ HashReturn Keccak_HashInitialize(Keccak_HashInstance *hashInstance, unsigned int

/** Macro to initialize a SHA3-256 instance as specified in the FIPS 202 standard.
 */
-#define Keccak_HashInitialize_SHA3_256(hashInstance) Keccak_HashInitialize(hashInstance, 1088,  512, 256, 0x06)
+#define Keccak_HashInitialize_SHA3_256(hashInstance) Keccak_HashInitialize(hashInstance, 1088,  512, 256, 0x01)

/** Macro to initialize a SHA3-384 instance as specified in the FIPS 202 standard.
 */
@@ -78,7 +78,7 @@ HashReturn Keccak_HashInitialize(Keccak_HashInstance *hashInstance, unsigned int

/** Macro to initialize a SHA3-512 instance as specified in the FIPS 202 standard.
 */
-#define Keccak_HashInitialize_SHA3_512(hashInstance) Keccak_HashInitialize(hashInstance,  576, 1024, 512, 0x06)
```

```
+#define Keccak_HashInitialize_SHA3_512(hashInstance)
Keccak_HashInitialize(hashInstance,  576, 1024, 512, 0x01)

/**
 * Function to give input data to be absorbed.
```

Figure E.2: Changes needed to turn the SHA-3 reference implementations into Keccak

F. Arithmetic Instruction Fuzzing Code

Figure F.1 contains the code used to fuzz the new arithmetic instructions introduced by the TVM upgrade.

The code in figure F.1 works roughly as follows. It reads five integers from standard input. Those five integers are used to construct Fift code. The Fift code exercises one of the new instructions, as chosen by the fifth integer read from standard input. Some subset of the first four integers are used as arguments to the new instruction. The result returned by the instruction is compared to values computed outside of, and embedded in, the Fift code. Examples of such generated Fift code appear in figures 11.5 and 15.3.

AFLplusplus was the fuzzing engine used to run the code in figure F.1. The code was run with a trivial (essentially meaningless) initial corpus.

```
#include <err.h>
#include <unistd.h>

#include "common/bigint.hpp"
#include "fift/utils.h"
#include "vm/vm.h"

using namespace fift;
using namespace td;
using namespace vm;

struct CodeQR {
  const char *code;
  RefInt256 q;
  RefInt256 r;
  bool needs_y_early;
  bool needs_z;
  bool needs_y_late;
};

struct CodeQRInner {
  const char *code;
  RefInt256 dividend;
  RefInt256 divisor;
  bool needs_y_early;
  bool needs_z;
  bool needs_y_late;
};

struct FloorRoundCeil {
  RefInt256 a[3];
};

#define bail(msg)     \
```

```
  do {                    \
    printf(msg "\n");  \
    exit(0);           \
  } while (0)

#define opts(cond, s) ((cond) ? (s) : "")

CodeQR compute(const RefInt256 &x, const RefInt256 &y, const RefInt256 &w, const RefInt256 &z, unsigned op);
CodeQRInner compute_inner(const RefInt256 &x, const RefInt256 &y, const RefInt256 &w, const RefInt256 &z,
                          unsigned base_op);
long checked_to_long(const RefInt256 &x);
FloorRoundCeil floor_round_ceil(const RefInt256 &x, const RefInt256 &y);

int main() {
  char *xs = nullptr, *ys = nullptr, *ws = nullptr, *zs = nullptr;
  char *fift = nullptr;

  try {
    unsigned op;
    if (scanf("%ms %ms %ms %ms %u", &xs, &ys, &ws, &zs, &op) < 5) {
      bail("too few args");
    }

    BigInt256 x, y, w, z;
    if (strlen(xs) == 0 || x.parse_dec(xs) != strlen(xs) || !x.signed_fits_bits(257)) {
      bail("bad x");
    }
    if (strlen(ys) == 0 || y.parse_dec(ys) != strlen(ys) || !y.signed_fits_bits(257)) {
      bail("bad y");
    }
    if (strlen(ws) == 0 || w.parse_dec(ws) != strlen(ws) || !w.signed_fits_bits(257)) {
      bail("bad w");
    }
    if (strlen(zs) == 0 || z.parse_dec(zs) != strlen(zs) || !z.signed_fits_bits(257)) {
      bail("bad z");
    }
    if (op >= 24) {
      bail("bad op");
    }

    printf("x  = %s\n", xs);
    printf("y  = %s\n", ys);
    printf("w  = %s\n", ws);
    printf("z  = %s\n", zs);
    printf("op = %u\n", op);

```

```
        CodeQR code_qr = compute(make_refint(x), make_refint(y), make_refint(w), make_refint(z), op);

        if (!code_qr.q->is_valid()) {
          bail("q is NaN");
        }

        if (!code_qr.r->is_valid()) {
          bail("r is NaN");
        }

        bool q_overflow = !code_qr.q->signed_fits_bits(257);
        bool r_overflow = !code_qr.r->signed_fits_bits(257);

        string qs = code_qr.q->to_dec_string();
        string rs = code_qr.r->to_dec_string();

        asprintf(&fift,
                 "\n\
                  { \n\
                    =: ans-r =: ans-q \n\
                    %s =: w %s =: x \n\
                    @' x %s @' w %s %s \n\
                    <b x{%s} s, b> <s 0 runvmx \n\
                    .s \n\
                    abort\"Exitcode != 0\" \n\
                    @' ans-r <> abort\"Incorrect r\" \n\
                   @' ans-q <> abort\"Incorrect q\" \n\
                  } : test \n\
                  \n\
                  %s %s %s %s %s %s test",
                 opts(code_qr.needs_z, "=: z"), opts(code_qr.needs_y_early || code_qr.needs_y_late, "=: y"),
                 opts(code_qr.needs_y_early, "@' y"), opts(code_qr.needs_z, "@' z"), opts(code_qr.needs_y_late, "@' y"),
                 code_qr.code, xs, opts(code_qr.needs_y_early || code_qr.needs_y_late, ys), ws, opts(code_qr.needs_z, zs),
                 qs.c_str(), rs.c_str());

        printf("%s\n", fift);

        auto res = mem_run_fift(fift);

        if (res.is_error()) {
          auto s = res.error().to_string();
          printf("%s\n", s.c_str());
          bool integer_overflow = strlen(s.c_str()) >= 4 && strcmp(s.c_str() + strlen(s.c_str()) - 4, ":-?]") == 0;
          assert((q_overflow || r_overflow) == integer_overflow);
          if (integer_overflow) {
            goto out;
          }
        }
```

```

    assert(!(q_overflow || r_overflow));

    res.ensure();
  } catch (VmError &err) {
    printf("%s\n", err.get_msg());
    throw;
  }
out:

  free(xs);
  free(ys);
  free(ws);
  free(zs);
  free(fift);

  return 0;
}

CodeQR compute(const RefInt256 &x, const RefInt256 &y, const RefInt256 &w, const RefInt256 &z, unsigned op) {
  static char code[7] = "xxxxxx";

  const unsigned base_op = op / 3;
  const unsigned round_mode = op % 3;

  CodeQRInner code_qr_inner = compute_inner(x, y, w, z, base_op);

  strcpy(code, code_qr_inner.code);
  code[3] += round_mode;
  RefInt256 q = floor_round_ceil(code_qr_inner.dividend, code_qr_inner.divisor).a[round_mode];
  RefInt256 r = code_qr_inner.dividend - q * code_qr_inner.divisor;
  return {code, q, r, code_qr_inner.needs_y_early, code_qr_inner.needs_z, code_qr_inner.needs_y_late};
}

CodeQRInner compute_inner(const RefInt256 &x, const RefInt256 &y, const RefInt256 &w, const RefInt256 &z,
                          unsigned base_op) {
  static char code[7] = "xxxxxx";
  switch (base_op) {
    case 0: {
      // MULADDDIVMOD    x y w z - q=floor((xy+w)/z) r=(xy+w)-zq
      return {
          "A980", x * y + w, z, true, true,
      };
    }
    case 1: {
      // ADDDIVMOD x w z - q=floor((x+w)/z) r=(x+w)-zq
      return {
          "A900", x + w, z, false, true,
      };
```

```
        }
        case 2: {
          // ADDRSHIFTMOD     x w z - q=floor((x+w)/2^z) r=(x+w)-q*2^z
          long z_long = checked_to_long(z);
          return {
              "A920", x + w, make_refint(1) << z_long, false, true, false,
          };
        }
        case 3: {
          // z ADDRSHIFT#MOD  x w - q=floor((x+w)/2^z) r=(x+w)-q*2^z
          long z_long = checked_to_long(z);
          if (z_long == 0) {
            bail("zero z");
          }
          sprintf(code, "A930%02lX", (z_long - 1) & 0xff);
          return {
              code, x + w, make_refint(1) << z_long, false, false, false,
          };
        }
        case 4: {
          // MULADDRSHIFTMOD  x y w z - q=floor((xy+w)/2^z) r=(xy+w)-q*2^z
          long z_long = checked_to_long(z);
          return {
              "A9A0", x * y + w, make_refint(1) << z_long, true, true, false,
          };
        }
        case 5: {
          // z MULADDRSHIFT#MOD     x y w - q=floor((xy+w)/2^z) r=(xy+w)-q*2^z
          long z_long = checked_to_long(z);
          if (z_long == 0) {
            bail("zero z");
          }
          sprintf(code, "A9B0%02lX", (z_long - 1) & 0xff);
          return {
              code, x * y + w, make_refint(1) << z_long, true, false,
          };
        }
        case 6: {
          // LSHIFTADDDIVMOD  x w z y - q=floor((x*2^y+w)/z) r=(x*2^y+w)-zq
          long y_long = checked_to_long(y);
          return {
              "A9C0", x * (make_refint(1) << y_long) + w, z, false, true, true,
          };
        }
        case 7: {
          // y LSHIFT#ADDDIVMOD     x w z - q=floor((x*2^y+w)/z) r=(x*2^y+w)-zq
          long y_long = checked_to_long(y);
          if (y_long == 0) {
            bail("zero y");
          }
          sprintf(code, "A9D0%02lX", (y_long - 1) & 0xff);
          return {
              code, x * (make_refint(1) << y_long) + w, z, false, true, false,
```

```
        };
      }
      default: {
        assert(false);
      }
    }
}

long checked_to_long(const RefInt256 &x) {
  long x_long = x->to_long();
  if (x < 0 || x > 256) {
    bail("bad x");
  }
  return x_long;
}

FloorRoundCeil floor_round_ceil(const RefInt256 &x, const RefInt256 &y) {
  int one_toward_zero = y < 0 ? 1 : -1;
  int y_sgn = y->sgn();
  return {
      x / y,
      (x + (y * y_sgn / make_refint(2)) * y_sgn) / y,
      (x + (y + one_toward_zero)) / y,
  };
}
```

Figure F.1: The code used to fuzz the new arithmetic instructions

G. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From October 23 to October 25, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the TON team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 22 issues described in this report, TON resolved 1 and partially resolved five. The status of one fix is undetermined. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Inadequate testing	Informational	Partially Resolved
2	Insufficient code comments	Informational	Partially Resolved
3	Hash bit ordering differs from FIPS 202	Informational	Resolved
4	Action phase fines can be bypassed	Undetermined	Undetermined
5	Use of deprecated OpenSSL APIs	Informational	Partially Resolved
6	MULADDDIVMOD and related instructions have unclear behavior	Informational	Resolved
7	Undefined behavior in CyclicBlobViewImpl	High	Resolved
8	Use of blst version with new-delete mismatch	High	Resolved
9	Arithmetic opcodes handled inconsistently	Informational	Partially Resolved

10	Inconsistencies between arithmetic operations' implementation and specification	Low	Resolved
11	Missing call to normalize in ADDDIVMOD implementation	High	Resolved
12	Use of deprecated cryptographic APIs	Informational	Resolved
13	Bignum can segfault when converting to string or hex	Informational	Resolved
14	Risk of infinite loop during RaptorQ FEC	Undetermined	Resolved
15	Missing to call to normalize in MULADDRSHIFT#MOD implementation	High	Resolved
16	BLS gas costs are inconsistent with specification	Low	Resolved
17	Use of libsodium might stall the process	Low	Resolved
18	RIST255_MUL uses nonstandard method for handling errors	High	Resolved
19	Cell slices for public keys and signatures can have excess data	Low	Partially Resolved
20	Divergent behavior among BLS instructions when n is 0	Informational	Resolved
21	Uninitialized data read when downcast_call fails	High	Resolved
22	Register c7 tuple element "previous blocks" can be null	Undetermined	Resolved

Detailed Fix Review Results

TOB-TVMUP-1: Inadequate testing
Partially resolved in commits 5a6ad4e513b06187ae91b27cf3b54d2f95e5da3d and 5fa20f46c3aa32a320ca527d5131b3b8d38e85e7. The TON team fixed existing tests, removed obsolete tests, and integrated tests into the GitHub CI pipeline.

TOB-TVMUP-2: Insufficient code comments
Partially resolved in commit 2088ee57c53f619082b228fbe22bb8a051621aff. The TON team added documentation to transaction.cpp, collator.cpp, and validate-query.cpp.

TOB-TVMUP-3: Hash bit ordering differs from FIPS 202
Resolved in commit aacbf3e0dd86932e1192e0b6d7b5cfc7d45b7afc. The TON team updated the documentation. Note, however, that the new documentation does not explicitly state that the TON behavior differs from FIPS; we recommend that this information be added.

TOB-TVMUP-4: Action phase fines can be bypassed
Undetermined. The client provided the following context for this finding's fix status:

> *Action fine for the current message is limited by msg_balance_remaining - gas_fees - ap.action_fine (see max_cells), so fine cannot exceed remaining balance.*

This implies that the exploit scenario provided for this finding is impossible because a check elsewhere in the code prevents the edge case. We were unable to confirm this either during the scope of the fix review or the original assessment, hence the undetermined severity of this finding and the undetermined rating for this finding's fix status.

TOB-TVMUP-5: Use of deprecated OpenSSL APIs
Partially resolved in commit 35fd778ff7bc4735f77187b8158971c958c35ed8. The TON team marked the MD5 function as deprecated but did not remove it from the codebase. The OpenSSL code does not yet use the EVP interface.

TOB-TVMUP-6: MULADDDIVMOD and related instructions have unclear behavior
Resolved in commit aacbf3e0dd86932e1192e0b6d7b5cfc7d45b7afc. The TON team updated the documentation.

TOB-TVMUP-7: Undefined behavior in CyclicBlobViewImpl
Resolved in commit d1a67b231b2aa6cb47c28f80c398fd497f15fcb3. The TON team added a check to ensure that the undefined behavior will not occur.

TOB-TVMUP-8: Use of blst version with new-delete mismatch
Resolved in commit b5ca7398c95888f1c1cf98aa25f2d3ded1abbbe6. The TON team

updated the `blst` dependency to version v0.3.11, which no longer has the `new-delete` mismatch vulnerability.

TOB-TVMUP-9: Arithmetic opcodes handled inconsistently
Partially resolved in commit `72357da63be4ee7a1cf5aac755eea2630334a0ab`. The TON team added the missing quiet opcodes, but there is still insufficient test coverage. The `QADDMULDIVMOD` opcode was also removed from `Asm.fif`.

TOB-TVMUP-10: Inconsistencies between arithmetic operations' implementation and specification
Resolved in commit `aacbf3e0dd86932e1192e0b6d7b5cfc7d45b7afc`. The TON team updated the documentation to match the implementation.

TOB-TVMUP-11: Missing call to normalize in ADDDIVMOD implementation
Resolved in commit `7efc1f6cfb0bfaa5804305e1890ed9a40b71b9cc`. The TON team added the missing call to `normalize`.

TOB-TVMUP-12: Use of deprecated cryptographic APIs
Resolved in commit `35fd778ff7bc4735f77187b8158971c958c35ed8`. The TON team marked the related code as deprecated.

TOB-TVMUP-13: Bignum can segfault when converting to string or hex
Resolved in commit `89bcfe7fde1bc0af9b19dcbbe5b91c5aaed511a9`. The TON team added a check to prevent the segfault.

TOB-TVMUP-14: Risk of infinite loop during RaptorQ FEC
Resolved in commit `6e654cfd2f6ceb8b4b403cdc67e574192014d0da`. The TON team changed the loop iterator type to `uint32`.

TOB-TVMUP-15: Missing to call to normalize in MULADDRSHIFT#MOD implementation
Resolved in commit `e3d230a6844f12e0c0d19665ef4f47beb01bd283`. The TON team added the missing call to `normalize`.

TOB-TVMUP-16: BLS gas costs are inconsistent with specification
Resolved in commit `aacbf3e0dd86932e1192e0b6d7b5cfc7d45b7afc`. The TON team updated the documentation to match the gas costs in the implementation.

TOB-TVMUP-17: Use of libsodium might stall the process
Resolved in commit `446d8c940b5427cc526e41cca059770516781c30`. The TON team refactored the way `libsodium` is initialized such that it now occurs when the TVM boots.

TOB-TVMUP-18: RIST255_MUL uses nonstandard method for handling errors
Resolved in commit `9d9c2e3106e5e0cb9a31491aaa6fd77019273659`. The TON team replaced the CHECK with a prior sign test.

TOB-TVMUP-19: Cell slices for public keys and signatures can have excess data
Partially resolved in commit `aacbf3e0dd86932e1192e0b6d7b5cfc7d45b7afc`. The documentation was updated to state that excess data is ignored. However, it is still possible for cell slices to have excess data and not throw an exception.

TOB-TVMUP-20: Divergent behavior among BLS instructions when n is 0
Resolved in commit `aacbf3e0dd86932e1192e0b6d7b5cfc7d45b7afc`. The TON team updated the documentation to match the implementation.

TOB-TVMUP-21: Uninitialized data read when downcast_call fails
Resolved in commit `5c1f48cf57304ca9fc3459cf5bd45e7a9f2c01c0`. The TON team added initial values to the stack variables as a precaution.

TOB-TVMUP-22: Register c7 tuple element "previous blocks" can be null
Resolved in commit f74f08645cc6eaff8181d590ed81aec58b3926e5. The TON team added source code documentation explaining the circumstances under which the previous blocks can be null.