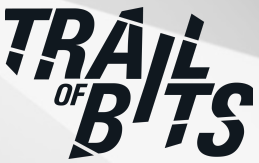

TON Blockchain

Final Report

April 12, 2023

Prepared for:
Justin Hyun, Head of Incubation
TON Foundation

Prepared by: **Henrik Brodin**, **Felipe Manzano**, and **Evan Sultanik**

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 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Executive Summary

Engagement Overview

TON Foundation engaged Trail of Bits to review the security of its TON blockchain. This consisted of reviews of the TON Virtual Machine (TVM), Fift scripting language, FunC smart contract programming language, Catchain consensus protocol, election contract, and smart contract bridge. From July 5 to October 28, 2022, a team of three consultants conducted a security review of the client-provided source code, with 24 person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code, documentation, and a test network. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	13
Medium	5
Low	17
Informational	12
Undetermined	4

CATEGORY BREAKDOWN

Category	*Count*
Data Exposure	1
Data Validation	16
Denial of Service	6
Error Reporting	4
Timing	3
Undefined Behavior	21

Notable Findings

The majority of findings are related to undefined behavior introduced by bugs in the C++ codebase, as well as lack of data validation. Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- Findings **TOB-TON-6**, **7**, **8**, and **9** are all related to undefined behavior in various TVM components that could lead to nondeterminism in the VM or even crashes due to crafted TVM opcode sequences.
- Findings **TOB-TON-3**, **21**, **22**, and **23** could all result in undefined behavior in a TON node, causing, at a minimum, denial of service.
- Findings **TOB-TON-30**, **36**, **45**, **46**, and **47** are all related to correct FunC code that will compile to semantically incorrect TVM code—i.e., FunC code that will behave differently than how the programmer specified.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager (First Half)
sam.greenup@trailofbits.com

Anne Marie Barry, Project Manager (Second Half)
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Henrik Brodin, Consultant
henrik.brodin@trailofbits.com

Felipe Manzano, Consultant
felipe.manzano@trailofbits.com

Evan Sultanik, Consultant
evan.sultanik@trailofbits.com

Project Timeline

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

Date	Event
June 30, 2022	Phase I (TVM and Fift) kickoff call
July 11, 2022	Status report #1
July 18, 2022	Status report #2
July 25, 2022	Status report #3
July 29, 2022	Status report #4
August 15, 2022	Phase II (Consensus) kickoff call
August 22, 2022	Status report #5
August 29, 2022	Status report #6
September 6, 2022	Status report #7
September 12, 2022	Status report #8

September 22, 2022	Phase III (FunC and the Bridge) kickoff call
October 3, 2022	Status update meeting #9
October 11, 2022	Status update meeting #10
October 14, 2022	Bridge code furnished to Trail of Bits
October 24, 2022	Status update meeting #11
October 31, 2022	Delivery of draft final report
October 31, 2022	Final status update
January 10, 2023	Delivery of final report
March 27, 2023	Fix review commenced
April 12, 2023	Delivery of fix review

Project Goals

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

- Can a maliciously crafted TVM bytecode program or Fift script cause a node to crash?
- Can a maliciously crafted TVM bytecode program, Fift script, or FunC contract cause a node to expend more computational resources than the gas cost?
- Are TVM programs, Fift scripts, and FunC contracts deterministic?
- Can a maliciously crafted TVM bytecode program, Fift script, or FunC contract be exploited to gain arbitrary code execution?
- Is the bytecode resulting from the compilation of FunC contracts semantically equivalent regardless of optimization level?
- Are the cryptographic primitives sound?
- Can a malicious node or minority coalition of nodes perform a denial-of-service attack on the network?
- Are there any flaws in the bridge contracts that would allow an attacker to freeze or steal funds?

Project Targets

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

TON Monorepo Containing Fift and the TVM

Repository	https://github.com/ton-blockchain/ton/
Version	`eb86234a1120fc3f9c6b390f4471cfd92b875044`
Type	Smart Contract Virtual Machine and Programming Language
Platform	C++

TON Monorepo Containing the Catchain and Validator Implementations

Repository	https://github.com/ton-blockchain/ton/
Version	`36fbe3a2acda90fb92826b114e71ac08a8e53438`
Type	Consensus Protocol and Blockchain Verifier
Platform	C++

TON Monorepo Containing the FunC Compiler

Repository	https://github.com/ton-blockchain/ton/
Version	`4b940f8bad9c2d3bf44f196f6995963c7cee9cc3`
Type	FunC Compiler
Platform	C++, FunC, and Fift

TON Bridge FunC Contracts

Repository	https://github.com/ton-blockchain/bridge-func
Version	`d03dbdbe9236e01efe7f5d344831bf770ac4c613`
Type	FunC Smart Contracts
Platform	FunC and Fift

TON Bridge Solidity Contracts

Repository	https://github.com/ton-blockchain/bridge-solidity
Version	`c5f51c1f40620ca3473788a203a387caed1e0897`
Type	Solidity Smart Contracts
Platform	Solidity

Project Coverage

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

- Static analysis of the entire TON monorepo
- Manual review of the TVM, Fift interpreter, FunC compiler, Catchain consensus protocol, validator, election contract, and smart contract bridge
- Fuzz testing of the bag of cells data structure, TVM opcodes, and FunC compiler
- Differential testing of the FunC compiler
- Opcode benchmarking (CPU time versus gas cost)
- In vivo testing via MyLocalTon
- Evaluation of clock drifting robustness using MyLocalTon with custom binaries
- Verification of serialization/deserialization of BlockSignatureSet
- FunC code and test case generation to identify incorrect code generation

Coverage Limitations

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

- The TVM has thousands of unique opcode variants, of which we were able to test only a small fraction. We observed that many opcodes of the same family have different runtimes depending on their constant arguments (see TOB-TON-11). We have included our test code in Appendix H. Benchmarking of the entire set of opcodes would be beneficial.
- The codebase would benefit from additional fuzz test harnesses (e.g., in the validator and FunC smart contract compiler). See Appendix F.
- Several findings have the potential to be high severity, but are currently classified with undetermined severity because there was insufficient time to confirm that they are exploitable with a proof of concept.
- Due to a delay in receiving the final version of the code, the bridge contracts were assessed only during the last calendar week of the assessment.

Automated Testing

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

Test Harness Configuration

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

Tool	Description	Policy
Cppcheck	Cppcheck is a static analysis tool for C/C++ code. It provides unique code analysis to detect bugs and focuses on detecting undefined behavior and dangerous coding constructs.	Appendix E
LLVM Sanitizers	Compile-time passes that add instrumentation to detect address misuse (ASAN), memory safety issues (MSAN), and undefined behavior (UBSAN) at runtime.	Appendix D and Appendix F
LibFuzzer	An in-process, coverage-guided, evolutionary fuzzing engine. LibFuzzer can automatically generate a set of inputs that exercise as many code paths in the program as possible.	Appendix F
test-timing	A custom utility that benchmarks TVM opcodes and compares their CPU usage against their gas cost.	Appendix H
FunC differential testing	Tool that constructs FunC expressions and evaluates equality for different optimization levels.	Appendix F
FunC model verification	Tool that constructs FunC expressions and evaluates equality to a Python-based model.	Appendix F

Slither	Static analyzer that scans Ethereum smart contracts for known-vulnerable patterns. https://github.com/crytic/slither	Public and Proprietary Vulnerability Detectors

Areas of Focus

Our automated testing and verification work focused on the following system properties:

- The program does not access invalid memory addresses.
- The program does not exercise undefined behavior.
- TVM opcodes consume computational resources proportional to their gas costs.
- The Ethereum bridge contracts do not contain any known-vulnerable patterns.

Test Results

The results of this focused testing are detailed below.

Property	Tool	Result
BagOfCell_deserialize. Randomly generated data fed to `Vm::BagOfCell::deserialize()` will not trigger memory safety, undefined behavior, or abrupt termination errors.	LibFuzzer	**Passed**
run_vm_code. Randomly generated cells fed to `Vm::run_vm_code` will not trigger memory safety, undefined behavior, or abrupt termination errors.	LibFuzzer	**TOB-TON-6, 7, 8, and 9**
run_vm_code_specific. Randomly generated cells containing valid instructions fed to `Vm::run_vm_code` will not trigger memory safety, undefined behavior, or abrupt termination errors.	LibFuzzer	**Passed**
Test-timing. TVM opcodes consume computational resources commensurate with their gas cost.	test-timing	**TOB-TON-11**

FunC differential testing. Randomly created expressions evaluate to equal results regardless of optimization level.	FunC differential testing	TOB-TON-30
FunC correctness testing. Randomly created expressions evaluate to the result of a Python-based model.	FunC model verification	TOB-TON-32, 36, 37, 38, 42, 43, and 47
Ethereum bridge contracts. Tested the Ethereum smart contracts for known vulnerable patterns using both Trail of Bits' public and proprietary detectors.	Slither	Passed

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	Several high-severity findings related to arithmetic errors: improper bit shifting of negative values and signed integer overflow.	**Weak**
Auditing	The TVM and Fift have robust logging and debugging capabilities. However, a few findings (TOB-TON-33, 34, 35, and 53) relate to inadequate or missing FunC compiler warnings or errors.	**Moderate**
Authentication / Access Controls	The primary component of the system with authentication and access controls assessed in the scope of this engagement was the bridge. This was assessed for only one week, resulting in no findings related to authentication. Anyone with access to the TON GitHub organization or repositories could surreptitiously introduce malicious changes into the codebase. The access controls surrounding TON's GitHub infrastructure were not assessed during this engagement.	**Further Investigation Required**
Complexity Management	Although the codebase is well organized, a lack of inline documentation and IDE's inability to resolve virtual methods in all contexts sometimes hindered manual code review. However, we identified several utilities required to successfully develop with TON (e.g., MyLocalTon and `toncli`) distributed across unofficial, personal GitHub repositories.	**Moderate**

Configuration	The TVM has many configuration options. Discovering the purpose of these options and/or the units of their values was often possible only by inspecting the code.	**Moderate**
Cryptography and Key Management	We did not discover any cryptographic flaws in the system; however, we recommend against allowing pseudorandom numbers to be generated on-chain (see TOB-TON-13). Several deprecated cryptographic functions from the OpenSSL library are used, but they do not appear to be exploitable (see Appendix C).	**Satisfactory**
Data Handling	Several inputs were discovered that could cause the func compiler to crash; however, these did not result in high-severity security issues. The system as a whole could be improved by including property-based or fuzz tests to exercise unintended inputs (see Appendix F).	**Satisfactory**
Decentralization	Several findings could produce nondeterminism in the TVM, leading to consensus issues (e.g., TOB-TON-24). However, these findings were not directly exploitable.	**Satisfactory**
Documentation	The high-level documentation about the TON blockchain, the TVM, and Fift is excellent. However, the codebase could benefit from more inline comments. The documentation on the FunC language is also imprecise. For example, it is unclear whether mixed bit-length dictionary operations are intended (see TOB-TON-45, 46, and 49).	**Moderate**
Maintenance	This assessment did not assess the maintenance of the codebase or its deployments.	**Not Considered**
Memory Safety and Error Handling	Several findings resulted from memory safety errors. Although tools such as the Address Sanitizer (asan) could detect many of these errors, TON cannot be built with asan enabled without disabling some checks (see the General Recommendations section of Appendix C).	**Weak**

Testing and Verification	Some TVM opcodes have no unit test coverage. There is no comprehensive testing of the correctness of Fift code emitted by the FunC compiler. There are no integrated testing harnesses in the TON repository itself, preventing testing the system in a simulated network with the consensus protocol running. The codebase has no automated property-based testing, fuzz testing, or formal verification. We recommend updating the codebase to resolve all issues that lead to current compiler warnings, or to suppress warnings that are known to be false positives (see Appendix G).	**Weak**

Summary of Findings

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

ID	Title	Type	Severity
1	Proxied ADNL pong messages may have empty data	Undefined Behavior	Informational
2	A block ID with no associated queue will cause a crash	Denial of Service	Informational
3	Token manager only checks every other download for timeouts	Denial of Service	High
4	FunC compiler will dereference an invalid pointer when output file is provided	Denial of Service	Low
5	ListIterator postfix increment operator returns a local variable by reference	Undefined Behavior	Undetermined
6	TVM programs can trigger undefined behavior in bigint.hpp	Undefined Behavior	High
7	TVM programs can trigger undefined behavior in bitstring.cpp	Undefined Behavior	High
8	TVM programs can trigger undefined behavior in tonops.cpp	Undefined Behavior	High
9	TVM programs can trigger undefined behavior in CellBuilder.cpp	Undefined Behavior	High
10	Multiple Fift stack instructions fail to check the stack depth	Undefined Behavior	Low
11	PUSHPOW2 opcode uses twice as much CPU time as opcodes with a similar gas cost	Denial of Service	Low

12	Stack use-after-scope in tdutils test	Undefined Behavior	**Informational**
13	On-chain pseudorandom number generation	Data Exposure	**Informational**
14	Retracted as a result of further investigation during the fix review *		**Undetermined**
15	VM state guards fail when not assigned to a variable	Timing	**Low**
16	Performance warning timers in the cell DB do not work	Timing	**Low**
17	DHT queries will crash if debug logging is enabled	Undefined Behavior	**Low**
18	Frequent connection state changes can cause an ADNL node to exhaust memory	Denial of Service	**Informational**
19	Missing base copy constructor invocation in derived copy constructor	Undefined Behavior	**Informational**
20	Unbounded storage of received Catchain blocks	Denial of Service	**Informational**
21	Getting account state can crash when building a state root proof	Undefined Behavior	**High**
22	Misaligned object allocation and interaction	Undefined Behavior	**High**
23	Use of DowncastHelper leads to invalid downcast of incorrect type	Undefined Behavior	**High**
24	Clock drift can break consensus	Timing	**Informational**
25	Shard records can be instantiated with uninitialized member variables	Undefined Behavior	**Undetermined**

26	Signatures of block antecessors are not validated	Data Validation	**Undetermined**
27	TLB reference validation can be bypassed	Data Validation	**Undetermined**
28	The TON client's get shards request can fail	Undefined Behavior	**Low**
29	Bigint and cell tests can silently fail due to undefined behavior	Undefined Behavior	**Low**
30	Multiplication of a constant can lead to a misaligned stack	Data Validation	**High**
31	FunC codegen invokes undefined behavior	Undefined Behavior	**Medium**
32	Constant operations on NaN can cause the FunC compiler to crash	Undefined Behavior	**Low**
33	Undefined variables in FunC are treated as undefined functions and do not cause a compiler error	Error Reporting	**Medium**
34	Calls to implicitly impure functions without a return value are always optimized out without an error	Error Reporting	**Medium**
35	Calls to implicitly impure functions with unused return values are always optimized out without an error	Error Reporting	**Informational**
36	Comparison to NaN results in the other comparand	Data Validation	**High**
37	FunC fails to reject out-of-range constants	Data Validation	**Low**
38	Inconsistent runtime behavior for operations resulting in NaN	Data Validation	**Medium**

39	Missing _Bit-marker for positive integer 1	Data Validation	**Informational**
40	Method IDs can collide without warning	Data Validation	**Low**
41	Single-line comments are honored within multi-line comments	Data Validation	**Low**
42	Bitwise operators can cause the FunC compiler to crash	Undefined Behavior	**Low**
43	FunC compiler can produce undefined opcodes	Undefined Behavior	**Low**
44	Invalid syntax can cause the FunC compiler to crash	Undefined Behavior	**Low**
45	Dictionary lookup can return incorrect results	Data Validation	**High**
46	Dictionary insertion can inconsistently crash	Data Validation	**High**
47	Bitwise negation of false is not always true	Data Validation	**High**
48	Setting the random number seed from the FunC standard library causes a stack misalignment	Data Validation	**Medium**
49	Querying a dictionary throws exception	Data Validation	**Low**
50	Compile time integer literal operations can result in unexpected control flow	Data Validation	**Low**
51	Retracted as a result of further investigation during the fix review *		**Undetermined**
52	Ethereum bridge signature verification will always pass for address zero	Data Validation	**Informational**

53	Context sensitivity of the ; token can lead to confusion and bugs	Error Reporting	**Informational**
54	Retracted as a result of further investigation during the fix review *		**Undetermined**

* These findings of undetermined severity had been previously reported in a provisional state. Further investigation during the fix review determined that these findings, as originally reported, were invalid. A discussion of the findings as well as relevant recommendations are included in appendix J. These entries remain as placeholders in order to preserve the previously reported finding IDs.

Detailed Findings

1. Proxied ADNL pong messages may have empty data	
Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-1
Target: `adnl/adnl-proxy.cpp`	

Description
The TON Abstract Datagram Network Layer (ADNL) protocol proxy incorrectly re-copies Pong control packet data to itself after having already been moved by `std::move`, as depicted in Figure 1.1. Although this is valid C++ code, after line 188 the `data` object will remain in an undefined state; the compiler has the option to erase its contents. Therefore, the second move on line 191 will potentially wipe `p.data`.

```
188    p.data = std::move(data);
189    p.adnl_start_time = start_time();
190    p.seqno = out_seqno_;
191    p.data = std::move(data);
```

Figure 1.1: Duplicate move of the contents of `data` in `adnl-proxy.cpp`

This finding is informational because Pong messages still function to keep a connection alive regardless of whether they contain a data payload.

Exploit Scenario
A TON node sends invalid Pong messages containing no data payload, causing the node to be disconnected from its peers.

Recommendations
Short term, remove the erroneous second move on line 191.

Long term, integrate linting tools like cppcheck or clang-tidy into your CI pipeline that can detect use-after-move bugs.

2. A block ID with no associated queue will cause a crash

Severity: **Informational**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-TON-2
Target: `crypto/block/block-db.cpp`	

Description

Obtaining queue information for an invalid block ID leads to an invalid iterator access. It appears that a `return` statement was intended but omitted between lines 668 and 669 of `block-db.cpp`:

```
666    if (it == state_info.end()) {
667      promise(td::Status::Error(
668          -666, std::string{"cannot obtain output queue info for block "} +
           blk_id.to_str() + " : cannot load state"));
669    }
670    if (it->second->data.is_null()) {
```

Figure 2.1: Missing return statement before line 670 results in an invalid iterator access

Since the error handling code inside the `if` block will fall through, the `it` iterator will be invalid when it is dereferenced on line 670, causing a segfault.

This finding is informational because it does not appear that the `BlockDbImpl::get_out_queue_info_by_id` function containing this bug is actually called anywhere in the code. However, if a code path that reaches this function exists, the severity of this finding would be high.

Exploit Scenario

A code path reaches this function to retrieve queue information for a block specified in an ADNL message. A malicious node crafts an ADNL message containing a nonexistent block ID, causing all of its peers to crash.

Recommendations

Short term, add a `return` statement between lines 668 and 669.

Long term, determine whether this code is actually used and, if not, consider removing it.

3. Token manager only checks every other download for timeouts

Severity: **High**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-TON-3
Target: `validator/token-manager.cpp`	

Description
Each actor's token manager periodically checks if its pending token downloads have timed out. The loop that iterates over the pending downloads follows in Figure 3.1.

```
70    for (auto it = pending_.begin(); it != pending_.end(); it++) {
71      if (it->second.timeout.is_in_past()) {
72        it->second.promise.set_error(td::Status::Error(ErrorCode::timeout,
          "timeout in wait download token"));
73        auto it2 = it++;
74        pending_.erase(it2);
75      } else {
76        it++;
77      }
78    }
```

Figure 3.1: The iterator will be incremented twice in each for loop, skipping every other entry.

Note that the iterator is incremented twice: once in the `for` loop on line 70, and again in each branch of the `if` statement on lines 73 and 76. If the last element in the `pending_` mapping is timed out, then the second iterator increment will proceed past the end of the mapping.

Exploit Scenario
A validator with a poor network connection has many token download timeouts. If the timeouts occur more frequently than the call to the promise cleanup loop from Figure 3.1, then the pending token download queue will have unbounded increase. The last pending token promise times out, incrementing the iterator past the end of the mapping, accessing invalid memory and causing the validator to crash.

Recommendations
Short term, remove the unnecessary increment in the `for` loop on line 70.

Long term, integrate linting tools like cppcheck or clang-tidy into your CI pipeline that can detect improper iterator incrementing.

4. FunC compiler will dereference an invalid pointer when output file is provided

Severity: **Low**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-TON-4
Target: `crypto/func/func.cpp`	

Description
The `func` program will dereference an uninitialized unique pointer if an output filename is provided rather than printing to STDOUT.

```
271    std::unique_ptr<std::fstream> fs;
272    if (!output_filename.empty()) {
273      fs = std::make_unique<std::fstream>(output_filename, fs->trunc | fs->out);
```

Figure 4.1: The unique pointer is dereferenced before being initialized.

On line 273, the fs pointer is dereferenced twice before it is initialized.

Exploit Scenario
The `func` utility is invoked automatically with a filename specified (e.g., in a contract verification app similar to Etherscan). The utility crashes due to the invalid pointer dereference.

Recommendations
Short term, remove the invalid dereferences.

Long term, add integration tests to your CI pipeline to test all arguments of the command-line interfaces.

5. ListIterator postfix increment operator returns a local variable by reference

Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-5
Target: `crypto/func/func.h`	

Description

`ListIterator` is a utility class that wraps C style arrays and makes them easily iterable. Its postfix increment operator returns a local variable by reference.

As shown below on line 503, a new stack variable is returned by reference, which produces undefined behavior in C++.

```
500    ListIterator& operator++(int) {
501      T* z = ptr;
502      ptr = ptr->next.get();
503      return ListIterator{z};
504    }
```

Figure 5.1: The return-by-reference value for the postfix increment operator is a local variable.

The severity of this issue is undetermined because we did not exhaustively evaluate all uses of `ListIterator` for vulnerability to this bug.

Exploit Scenario

A list iterator is postfix-incremented and assigned to a new variable. The resulting variable will be an invalid reference and likely segfault on any member access or operation.

Recommendations

Short term, change the return type of the postfix operator to be a value rather than a reference.

Long term, integrate linting tools like cppcheck or clang-tidy into your CI pipeline that can detect stale reference bugs.

6. TVM programs can trigger undefined behavior in bigint.hpp

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-6
Target: `crypto/common/bigint.hpp`	

Description

The sequences of TVM operations shown in figures 6.1–6.22 trigger undefined behavior in `crypto/common/bigint.hpp`.

Executing code with undefined behavior in C++ allows the compiler to emit any and all possible code. Although the program may seem to work as expected, results will often differ depending on factors such as compiler choice, options, and execution environment. For example, compilers will often silently optimize away code that it can prove could execute undefined behavior. Appendix D provides a more in-depth discussion of undefined behavior and provides real-world examples and our general recommendations for addressing it.

Examples of TVM code that triggers undefined behavior are provided below. Each example can be triggered by running the following:

```
echo '2 3 1 1 29 12 x{aabbccdd} runvmcode .s'
|UBSAN_OPTIONS=print_stacktrace=1:halt_on_error=1:abort_on_error=
1 crypto/fift -I ../crypto/fift/lib/ -i
```

where `aabbccdd` is replaced with the corresponding TVM code. Assuming Fift is built with Undefined Behavior Sanitizer (ubsan) support, the program terminates with an error indicating the undefined behavior.

On line 323, the computation of `x + Tr::Half` can trigger a signed integer overflow.

```
322    } else {
323     digits[0] = ((x + Tr::Half) & (Tr::Base - 1)) - Tr::Half;
324     digits[n++] = (x >> Tr::word_shift) + (digits[0] < 0);
325    }
```

Figure 6.1: Undefined behavior can be invoked on line 323.
(crypto/common/bigint.hpp#322–325)

```
762    auto dm = std::div(exponent, word_shift);
763    int k = dm.quot;
764    while (size() <= k) {
765     digits[inc_size()] = 0;
766    }
767    digits[k] += ((word_t)factor << dm.rem);
```

Figure 6.2: Undefined behavior can be invoked on line 767.

On line 767, the computation `(word_t)factor << dm.rem` triggers a left shift of negative value -1. TVM code to trigger: `762020a9a9`.

```
967    word_t hi = 0;
968    Tr::add_mul(&hi, &digits[i + j], yv, zp.digits[j]);
969    if (hi && hi != -1) {
970     return invalidate_bool();
971    }
972    digits[size() - 1] += (hi << word_shift);
```

Figure 6.3: Undefined behavior can be invoked on line 972.

On line 972, the computation `hi << word_shift` triggers a left shift of negative value -1. TVM code to trigger: `85f87ca87ca8`.

```
1008    word_t v = digits[size() - 1];
1009    if (size() >= 2) {
1010     if (v >= Tr::MaxDenorm) {
1011       return 1;
1012     } else if (v <= -Tr::MaxDenorm) {
1013       return -1;
1014     }
1015     int i = size() - 2;
1016     do {
1017       v <<= word_shift;
```

Figure 6.4: Undefined behavior can be invoked on line 1017.

On line 1017, the computation `hi << word_shift` triggers a left shift of negative value -8. TVM code to trigger: `c8cf37c8cf37e317a9de2e`.

```
1062    word_t v = digits[0] + (digits[1] << word_shift);  // approximation mod
2^64
```

Figure 6.5: Undefined behavior can be invoked on line 1062.

On line 1062, the computation `digits[0] + (digits[1] << word_shift)` triggers signed integer overflow because -1 + -9223372036854775808 cannot be represented in type 'long long'.
TVM code to trigger: `843ee5`.

```
1062    word_t v = digits[0] + (digits[1] << word_shift);  // approximation mod
2^64
```

Figure 6.6: Undefined behavior can be invoked on line 1062.

Also on line 1062, the computation `digits[0] + (digits[1] << word_shift)` performs a left shift of 4096 by 52 places, which cannot be represented in type 'long long'. TVM code to trigger: `76aeaeae`.

```
1133    v = -yp.digits[--yn];
1134    if (v >= Tr::MaxDenorm) {
1135     return 1;
1136    } else if (v <= -Tr::MaxDenorm) {
1137     return -1;
1138    }
1139    while (yn > xn) {
1140     v <<= word_shift;
```

Figure 6.7: Undefined behavior can be invoked on line 1140.

On line 1140, the computation `v <<= word_shift` triggers a left shift of negative value -1. TVM code to trigger: `68839ba909`.

```
1153    v <<= word_shift;
```

Figure 6.8: Undefined behavior can be invoked on line 1153.

On line 1153, the computation `v <<= word_shift` triggers a left shift of negative value -1. TVM code to trigger: `68839ba9d9a4`.

```
1354    digits[size() - 1] += (digits[size()] << word_shift);
```

Figure 6.9: Undefined behavior can be invoked on line 1354.

On line 1270, the computation `(z << word_shift)` triggers a left shift of a negative value.

```
1269    if (!z || z == -1) {
1270     digits[size() - 1] += (z << word_shift);
1271     return true;
1272    } else {
```

Figure 6.10: Undefined behavior can be invoked on line 1270. (crypto/common/bigint.hpp#1269–1272)

On line 1341, the computation of `hi << word_shift` can trigger a left shift of a negative value.

```
1341    digits[size() - 1] += (hi << word_shift);
```

Figure 6.11: Undefined behavior can be invoked on line 1341 (crypto/common/bigint.hpp#1341)

On line 1354, the computation `digits[size()] << word_shift` triggers a left shift of negative value -1.
TVM code to trigger: `85a0855fa9da0a`.

```
1458    word_t pow = ((word_t)1 << q);
1459    word_t v = digits[size() - 1] & (pow - 1);
```

Figure 6.12: Undefined behavior can be invoked on line 1459.

On line 1459, the computation `pow-1` triggers signed integer overflow because `-9223372036854775808 - 1` cannot be represented in type 'long long'.
TVM code to trigger: `74a93e3e`.

```
1626    digits[size() - 1] += (v << word_shift);
```

Figure 6.13: Undefined behavior can be invoked on line 1626.

On line 1488, the computation `v - (w << word_shift)` can result in signed integer overflow (subtracting a large negative number originating from `w << word_shift`).

```
1486    } else if (v >= Tr::Half && size() < max_size()) {
1487     word_t w = (((v >> (word_shift - 1)) + 1) >> 1);
1488     digits[size() - 1] = v - (w << word_shift);
1489     digits[inc_size()] = w;
1490     return true;
1491    } else {
```

Figure 6.14: Undefined behavior can be invoked on line 1488. (crypto/common/bigint.hpp#1486–1491)

On line 1626, the computation `v << word_shift` triggers a left shift of negative value -1.
TVM code to trigger: `7caaeb`.

```
1775    word_t q = digits[k];
1776    if (k > 0 && q > -Tr::MaxDenorm / 2) {
1777     q <<= word_shift;
```

Figure 6.15: Undefined behavior can be invoked on line 1777.

On line 1769, the computation `q <<= word_shift` can trigger a left shift of negative value -243.

```
1763    while (k > 0) {
1764     if (q >= Tr::MaxDenorm / 2) {
1765       return s + 1;
1766     } else if (q <= -Tr::MaxDenorm / 2) {
1767       return s;
1768     }
1769     q <<= word_shift;
1770     q += digits[--k];
1771    }
```

Figure 6.16: Undefined behavior can be invoked on line 1769 (crypto/common/bigint.hpp#1763–1771)

On line 1777, the computation `v << word_shift` triggers a left shift of negative value -32. TVM code to trigger: `85a0b7b602`.

```
1925    td::bitstring::bits_store_long_top(buff, offs, v << (64 - bits), bits);
```

Figure 6.17: Undefined behavior can be invoked on line 1925.

On line 1793, the computation of `q <<= word_shift` can trigger a left shift of a negative value.

```
1787    while (k > 0) {
1788     if (q >= Tr::MaxDenorm / 2) {
1789       return s;
1790     } else if (q <= -Tr::MaxDenorm / 2) {
1791       return s + 1;
1792     }
1793     q <<= word_shift;
1794     q += digits[--k];
1795    }
```

Figure 6.18: Undefined behavior can be invoked on line 1793. (crypto/common/bigint.hpp#1787–1795)

On line 1830, the computation `v += (digits[i] << k);` can trigger a left shift of a negative value.

```
1829    } else {
1830     v += (digits[i] << k);
1831     k += word_shift;
1832    }
```

Figure 6.19: Undefined behavior can be invoked on line 1830 (crypto/common/bigint.hpp#1829–1832)

On line 1881, the computation v += (digits[i] << k); can trigger a left shift of a negative value.

```
1880    } else {
1881     v += (digits[i] << k);
1882     k += word_shift;
1883    }
```

Figure 6.20: Undefined behavior can be invoked on line 1881 (crypto/common/bigint.hpp#1880–1883)

On line 1925, the computation `v << (64 - bits)` triggers a left shift of negative value -1.
TVM code to trigger: `c868a3fa03`.

```
2045    unsigned long long val = td::bitstring::bits_load_long_top(buff, offs,
bits);
2046    if (sgnd) {
2047     digits[0] = ((long long)val >> (64 - bits));
2048    } else {
2049     digits[0] = (val >> (64 - bits));
```

Figure 6.21: Undefined behavior can be invoked on line 2049.

On line 1966, the computation `v += (digits[i] << k);` can trigger a left shift of a negative value.

```
1965    } else {
1966     v += (digits[i] << k);
1967     k += word_shift;
1968    }
```

Figure 6.22: Undefined behavior can be invoked on line 1966 (crypto/common/bigint.hpp#1965–1968)

On line 2049, the computation `(val >> (64 - bits))` performs a right shift using shift exponent 64 which is too large for 64-bit type 'unsigned long long'.
TVM code to trigger: `ed45d0d712`.

Exploit Scenario
Blockchain nodes running user-supplied TVM code behave differently when the undefined behavior is triggered, causing the network to lose consensus. Because undefined behavior can be triggered for very short instruction sequences, the attack need not be intentional.

Recommendations
Short term, consider switching to unsigned types with defined behavior for overflow and shifts and ensure that any out-of-range value cannot be produced.

Long term, integrate fuzzing of TVM with Undefined Behavior Sanitizer enabled to detect undefined behavior.

7. TVM programs can trigger undefined behavior in bitstring.cpp

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-7
Target: `crypto/common/bitstring.cpp`	

Description

The sequences of TVM operations shown in figures 7.1–7.2 trigger undefined behavior in `crypto/common/bitstring.cpp`.

Executing code with undefined behavior in C++ implies anything can happen. Although the code may seem to work as expected, results can differ depending on any factor.

For example, when Fift is built with Undefined Behavior Sanitizer, the undefined behavior shown in figure 7.1 can be triggered by running the following:

```
echo 'x{c8cf903f3f3f3f} runvmcode .s' |
UBSAN_OPTIONS=print_stacktrace=1:halt_on_error=1:abort_on_error=1
crypto/fift -I ../crypto/fift/lib/ -i
```

```
193    if (b > 0) {
194     *to = (unsigned char)((*to & (0xff >> b)) | ((int)acc << (8 - b)));
195    }
```

Figure 7.1: Undefined behavior can be invoked on line 194 of `crypto/common/bitstring.cpp`.

On line 194, the computation `((int)acc << (8 - b))` performs a left shift of value 530554783 by 7 places. The result cannot be represented by type 'int'.
TVM code to trigger: `c8cf903f3f3f3f`.

Additionally, lines 304, 322, and 330 of the `bits_memscan` function can all cause undefined behavior by shifting a negative value. This can be triggered by running the `test-smartcont` unit test.

Furthermore, on line 508, undefined behavior caused by a shift of 64 bits on a 64-bit type can happen when `bits` is 0.

```
507    unsigned long long bits_load_ulong(ConstBitPtr from, unsigned bits) {
```

```
508     return bits_load_long_top(from, bits) >> (64 - bits);
509   }
```

Figure 7.2: Undefined behavior can be invoked on line 508 when `bits` is 0 (`crypto/common/bitstring.cpp#507–509`)

Exploit Scenario

Blockchain nodes running user-supplied TVM code behave differently when the undefined behavior is triggered, causing the network to lose consensus. Because undefined behavior can be triggered for very short instruction sequences, the attack need not be intentional.

Recommendations

Short term, consider switching to unsigned types with defined behavior for overflow and shifts and ensure that any out-of-range value cannot be produced.

Long term, integrate fuzzing of TVM with Undefined Behavior Sanitizer enabled to detect undefined behavior.

8. TVM programs can trigger undefined behavior in tonops.cpp

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-8
Target: `crypto/vm/tonops.cpp`	

Description
A sequence of TVM operations triggers undefined behavior in `crypto/vm/tonops.cpp`.

Executing code with undefined behavior in C++ implies anything can happen. Although the code may seem to work as expected, results can differ depending on any factor.

When Fift is build with Undefined Behavior Sanitizer, the below example can be triggered by running the following:

```
echo 'x{c8853dfa02} runvmcode .s' |
UBSAN_OPTIONS=print_stacktrace=1:halt_on_error=1:abort_on_error=1
crypto/fift -I ../crypto/fift/lib/ -i
```

```
475    auto x = stack.pop_int();
476    auto cbr = stack.pop_builder();
477    unsigned len = ((x->bit_size(sgnd) + 7) >> 3);
```

Figure 8.1: Undefined behavior can be invoked on line 477 of `crypto/vm/tonops.cpp`.

On line 477, the computation `(x->bit_size(sgnd) + 7)` performs operation 2147483647+7 which cannot be represented by type 'int'.
TVM code to trigger: `c8853dfa02`.

Exploit Scenario
Blockchain nodes, running user-supplied TVM code, behave differently when the undefined behavior is triggered, causing the network to lose consensus. Because undefined behavior can be triggered for very short instruction sequences, the attack need not be intentional.

Recommendations
Short term, consider switching to unsigned types with defined behavior for overflow and shifts and ensure that any out-of-range value cannot be produced.

Long term, integrate fuzzing of TVM with Undefined Behavior Sanitizer enabled to detect undefined behavior.

9. TVM programs can trigger undefined behavior in CellBuilder.cpp

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-9
Target: `crypto/vm/cells/CellBuilder.cpp`	

Description
A sequence of TVM operations trigger undefined behavior in `crypto/vm/cells/CellBuilder.cpp`.

Executing code with undefined behavior in C++ implies anything can happen. Although the code may seem to work as expected, results can differ depending on any factor.

When Fift is built with the Undefined Behavior Sanitizer, the below example can be triggered by running the following:

```
echo 'x{686fa1ed44d7395af43e} runvmcode .s' |
UBSAN_OPTIONS=print_stacktrace=1:halt_on_error=1:abort_on_error=1
crypto/fift -I ../crypto/fift/lib/ -i
```

```
337    CellBuilder& CellBuilder::store_long(long long val, unsigned val_bits) {
338     return store_long_top(val << (64 - val_bits), val_bits);
339    }
```

Figure 9.1: Undefined behavior can be invoked on line 338 of `crypto/vm/cells/CellBuilder.cpp`.

On line 338, the computation `val << (64 - val_bits)` performs a left shift of the negative value -2.

TVM code to trigger: `686fa1ed44d7395af43e`.

Exploit Scenario
Blockchain nodes running user-supplied TVM code behave differently when the undefined behavior is triggered, causing the network to lose consensus. Because undefined behavior can be triggered for very short instruction sequences, the attack need not be intentional.

Recommendations
Short term, consider switching to unsigned types with defined behavior for overflow and shifts and ensure that any out-of-range value cannot be produced.

Long term, integrate fuzzing of TVM with Undefined Behavior Sanitizer enabled to detect undefined behavior.

10. Multiple Fift stack instructions fail to check the stack depth

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-10
Target: `crypto/fift/words.cpp`	

Description

Fift is described as a multipurpose scripting language script, similar to Bash. It is therefore expected to gracefully handle unexpected input and error states. Certain stack manipulation methods in `crypto/fift/stack.hpp:348` do not include an implicit stack underflow check to bail out in an orderly manner, resulting in undefined behavior and ultimately a crash. Callers of the stack API, such as in `crypto/fift/words.cpp`, are responsible for checking if the stack has enough values.

Two Fift instructions fail to check the correct stack depth before being interpreted: EQV and EQV?. In these situations, an undefined behavior condition can be reached that causes the interpreter to crash and, at best, exit abruptly.

```
1309    void interpret_is_eqv(vm::Stack& stack) {
1310      auto y = stack.pop(), x = stack.pop();
1311      stack.push_bool(are_eqv(std::move(x), std::move(y)));
1312    }
1313
1314    void interpret_is_eq(vm::Stack& stack) {
1315      auto y = stack.pop(), x = stack.pop();
1316      stack.push_bool(x == y);
1317    }
```

Figure 10.1: Undefined behavior can be invoked because the stack size is unchecked for EQV and EQV. (crypto/fift/words.cpp#1309–1317)

```
$ echo eq? eq? eqv? | catchsegv ./crypto/fift -I./crypto/fift/lib/ -i
 ok
Segmentation fault (core dumped)
*** Segmentation fault
```

Figure 10.2: A reproducible EQV crash

Exploit Scenario

A Fift script in production contains code that does not properly check the stack depth, which causes the script to unexpectedly crash.

Recommendations

Short term, ensure that all uses of `pop()` in the Fift instruction handlers are interpreted on a stack of sufficient depth. Document the trust boundaries related to Fift scripts.

Long term, integrate fuzzing of Fift with the Undefined Behavior Sanitizer (ubsan) enabled to detect undefined behavior.

11. PUSHPOW2 opcode uses twice as much CPU time as opcodes with a similar gas cost

Severity: **Low**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-TON-11
Target: `crypto/vm/arithops.cpp`	

Description
The runtime of the PUSHPOW2 TVM opcode is not constant over all inputs. For example,

```
[0xDF + 1] PUSHPOW2
```

runs in 0.009ms, but

```
[0x35 + 1] PUSHPOW2
```

requires over twice as much CPU time at 0.021ms. Other opcodes that cost the same 26 gas as PUSHPOW2 run significantly faster. For example, the DIVMOD opcode requires about 0.006ms of CPU time.

Exploit Scenario
An attacker sends carefully crafted, low-gas transactions to the TON blockchain, causing validators to expend an inordinate amount of CPU time.

Recommendations
Short term, consider increasing the gas cost of the PUSHPOW2 opcode.

Long term, continually benchmark the CPU overhead of each opcode. The time constraints of this assessment have not permitted us to test every possible combination of opcode and stack state. We have included our test harness in Appendix H, which can be extended by TON to benchmark all opcodes.

12. Stack use-after-scope in tdutils test

Severity: Informational	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-12
Target: `tdutils/test/List.cpp`	

Description

On destruction of the test case in Figure 12.1 below, the destructors are run in reverse order. Therefore, `id` is destroyed before `threads`. At that point, a thread in `threads` could still be running with a reference to `id`.

```
171    TEST(Misc, TsListConcurrent) {
172      td::TsList<ListData> root;
173      td::vector<td::thread> threads;
174      std::atomic<td::uint64> id{0};
175      for (std::size_t i = 0; i < 4; i++) {
176        threads.emplace_back(
177            [&] { do_run_list_test<td::TsListNode<ListData>,
td::TsList<ListData>, td::TsListNode<ListData>>(root, id); });
178      }
179    }
```

Figure 12.1: The `id` variable will be destructed before `threads`. (`tdutils/test/List.cpp#171–179`)

Exploit Scenario

A thread increments `id` after it has been destructed. Since the memory is no longer associated with `id`, the increment will overwrite data for another object now occupying the same memory. This could lead to a crash or other undefined behavior.

Recommendations

Short term, reorder

```
td::vector<td::thread> threads;
std::atomic<td::uint64> id{0};
```

to become

```
std::atomic<td::uint64> id{0};
td::vector<td::thread> threads;
```

Long term, run all tests with the LLVM address sanitizer (asan) enabled.

13. On-chain pseudorandom number generation

Severity: Informational	Difficulty: **Low**
Type: Data Exposure	Finding ID: TOB-TON-13
Target: `crypto/vm/tonops.cpp`	

Description
The TVM includes several opcodes for generating pseudorandom numbers on-chain. Since the entire chain is public and the TVM itself is deterministic, it is possible to predict the next random value with high accuracy, even if the pseudorandom number generator is seeded by the current time or block parameters as a source of entropy. This weakness has been thoroughly studied in Ethereum smart contracts.

Exploit Scenario
A malicious user exploits a lottery contract by predicting the winning value.

Recommendations
Short term, thoroughly document the risks of randomness without an external oracle.

Long term, consider deprecating the opcodes related to random number generation.

15. VM state guards fail when not assigned to a variable

Severity: Low	Difficulty: **Low**
Type: Timing	Finding ID: TOB-TON-15
Target: `crypto/vm/vm.cpp`	

Description

The `VMStateInterface` guard class uses the C++ idiom of Resource Acquisition Is Instantiation (RAII) to control the scope and lifetime of a guard. This is used, for example, when loading a library in order to prevent charging for cell load operations:

```
 597    Ref<Cell> VmState::load_library(td::ConstBitPtr hash) {
 598      std::unique_ptr<VmStateInterface> tmp_ctx;
 599      // install temporary dummy vm state interface to prevent charging for cell
load operations during library lookup
 600      VmStateInterface::Guard(tmp_ctx.get());
 601      for (const auto& lib_collection : libraries) {
 602        auto lib = lookup_library_in(hash, lib_collection);
 603        if (lib.not_null()) {
 604          return lib;
 605        }
 606      }
 607      missing_library = hash;
 608      return {};
 609    }
```

Figure 15.1: The guard object on line 600 will be immediately destructed after it is instantiated. (`crypto/vm/vm.cpp#597–609`)

Note that on line 600, the guard is never assigned to a variable. In this case, the guard will be instantiated and immediately destructed before line 601 is executed. The example in Figure 15.2, the output of which is in Figure 15.3, verifies this behavior.

```
#include <iostream>
class Guard {
  public:
  int value;
  Guard(int val) : value(val) {
    std::cout << "Created " << value << std::endl;
  }
  ~Guard() {
    std::cout << "Destroyed " << value << std::endl;
  }
```

```
};

void correct() {
  std::cout << "Correct" << std::endl;
  std::cout << "Before guard" << std::endl;
  Guard guard(0);
  std::cout << "After guard" << std::endl;
}

void incorrect() {
  std::cout << "Incorrect" << std::endl;
  std::cout << "Before guard" << std::endl;
  Guard(1);
  std::cout << "After guard" << std::endl;
}

int main(int argc, char* argv[]) {
  correct();
  std::cout << "---" << std::endl;
  incorrect();
  return 0;
}
```

Figure 15.2: A minimal example demonstrating the scope of an unassigned guard instantiation

```
$ g++ -O3 -std=c++17 object_create_no_name.cpp && ./a.out
Correct
Before guard
Created 0
After guard
Destroyed 0
---
Incorrect
Before guard
Created 1
Destroyed 1
After guard
```

Figure 15.3: The output of the minimal example in Figure 15.2

Exploit Scenario

TON users are erroneously charged for cell operations when loading libraries.

Recommendations

Short term, assign the guard to a variable to ensure its scope lasts the entire function.

Long term, increase unit test coverage to check gas cost invariants.

16. Performance warning timers in the cell DB do not work

Severity: Low	Difficulty: **Low**
Type: Timing	Finding ID: TOB-TON-16
Target: `validator/db/celldb.cpp`	

Description

Similar to finding TOB-TON-15, the `td::PerfWarningTimer` class uses RAII to control the scope and lifetime of timers. There are several instances in the cell database code where the timer is never assigned to a variable, so it will be immediately destructed after instantiation. (Figures 15.2 and 15.3, above, exemplify why this is dangerous.)

For example, the timers on lines 94 and 197 of `celldb.cpp` are ineffective.

```
 93    void CellDbIn::store_cell(BlockIdExt block_id, td::Ref<vm::Cell> cell,
td::Promise<td::Ref<vm::DataCell>> promise) {
 94      td::PerfWarningTimer{"storecell", 0.1};
```

Figure 16.1: The timer instantiated on line 94 is ineffective.
(validator/db/celldb.cpp#93–94)

```
196    void CellDbIn::gc_cont2(BlockHandle handle) {
197      td::PerfWarningTimer{"gccell", 0.1};
```

Figure 16.2: The timer instantiated on line 197 is ineffective.
(validator/db/celldb.cpp#196–197)

Exploit Scenario

A performance regression or edge case in the cell database goes unnoticed because the `PerfWarningTimer` erroneously underestimates the runtime of the functions.

Recommendations

Short term, assign the timers to variables to ensure their scope lasts for the entire function.

Long term, add static analyses that can detect unassigned instantiations with no side effects to TON's CI pipeline.

17. DHT queries will crash if debug logging is enabled

Severity: Low	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-17
Target: `dht/dht-query.hpp`	

Description

On construction of `DhtQueryFindValue` and `DhtQueryFindNodes` (Figure 17.1), the base `DhtQuery` constructor is called (line 109). In that constructor (Figure 17.2), `add_nodes` is called (line 59), which in turn calls `get_name` (three times), which is a pure virtual function. Calling a pure virtual function during object construction is undefined behavior.

If the `DHT_EXTRA_DEBUG` flag is enabled, then the first line of `add_nodes` will explicitly call `get_name` and immediately abort (Figure 17.3).

```
 106    DhtQueryFindNodes(DhtKeyId key, DhtMember::PrintId print_id,
adnl::AdnlNodeIdShort src, DhtNodesList list,
 107                      td::uint32 k, td::uint32 a, DhtNode self, bool
client_only, td::actor::ActorId<DhtMember> node,
 108                      td::actor::ActorId<adnl::Adnl> adnl,
td::Promise<DhtNodesList> promise)
 109        : DhtQuery(key, print_id, src, std::move(list), k, a, std::move(self),
client_only, node, adnl)
 110        , promise_(std::move(promise)) {
 111    }
```

Figure 17.1: The `DhtQueryFindNodes` constructor calls `DhtQuery` (`dht/dht-query.hpp#106–111`)

```
 40    class DhtQuery : public td::actor::Actor {
 41     protected:
 42      DhtKeyId key_;
 43      DhtNode self_;
 44      bool client_only_;
 45
 46     public:
 47      DhtQuery(DhtKeyId key, DhtMember::PrintId print_id, adnl::AdnlNodeIdShort
src, DhtNodesList list, td::uint32 k,
 48               td::uint32 a, DhtNode self, bool client_only,
td::actor::ActorId<DhtMember> node,
 49               td::actor::ActorId<adnl::Adnl> adnl)
 50          : key_(key)
 51          , self_(std::move(self))
 52          , client_only_(client_only)
```

```
53          , print_id_(print_id)
54          , src_(src)
55          , k_(k)
56          , a_(a)
57          , node_(node)
58          , adnl_(adnl) {
59        add_nodes(std::move(list));
60      }
```

Figure 17.2: DhtQuery calls add_nodes on line 59. (dht/dht-query.hpp#40–60)

```
67    VLOG(DHT_EXTRA_DEBUG) << this << ": " << get_name() << " query: received " <<
list.size() << " new dht nodes";
```

Figure 17.3: add_nodes calls get_name, which is a pure virtual function. (dht/dht-query.cpp#67)

Exploit Scenario

A node is running with DHT_EXTRA_DEBUG enabled. Upon its first DHT query, the node will abruptly terminate due to an abort.

Recommendations

Short term, ensure that pure virtual functions are never called from a constructor.

Long term, add static analyses that can detect this error case to TON's CI pipeline.

18. Frequent connection state changes can cause an ADNL node to exhaust memory

Severity: Informational	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-TON-18
Target: `adnl/adnl-peer.cpp`	

Description

When the connection state of an ADNL peer changes, all of the pending messages are sent, as shown below.

```
 771    void AdnlPeerPairImpl::conn_change_state(AdnlConnectionIdShort id, bool
ready) {
 772      if (ready) {
 773        if (pending_messages_.size() > 0) {
 774          send_messages_in(std::move(pending_messages_), true);
 775        }
 776      }
 777    }
```

Figure 18.1: Pending messages are flushed on connection state changes. (adnl/adnl-peer.cpp#771–777)

The pending messages vector is passed using move semantics. The C++ standard does not specify that a vector must be cleared after having been moved, only that it remains in a valid (but unspecified) state. Therefore, the original content of `pending_messages_` vector could technically remain while the `send_messages_in` function is being executed.

The messages passed to `send_messages_in` are added back to `pending_messages_` on lines 250 to 252 of the code in Figure 18.2. Therefore, the pending messages vector could be duplicated.

```
 235    void AdnlPeerPairImpl::send_messages_in(std::vector<OutboundAdnlMessage>
messages, bool allow_postpone) {
```

```
236      for (td::int32 idx = 0; idx < 2; idx++) {

             ⋮

250          for (auto &m : messages) {
251            pending_messages_.push_back(std::move(m));
252          }
```

Figure 18.2: The messages are added back to the pending messages queue. (adnl/adnl-peer.cpp#235–252)

This finding is informational because our experiments suggest that the major compilers and C++ standard libraries implicitly clear a vector after it is moved. However, the C++ standard does not explicitly require this, and there are instances where it is not guaranteed.

Exploit Scenario

Alice builds a TON node with an implementation of C++ that does not guarantee that moved vectors are implicitly cleared. Bob repeatedly connects to and disconnects from Alice's node. Each time, Alice's node's pending messages vector doubles in size, eventually exhausting memory.

Recommendations

Short term, move the pending messages vector to a temporary vector and explicitly clear the pending messages before calling `send_messages_in` on line 774 of Figure 18.1.

Long term, consider reducing the use of `std::move` in the codebase (see TOB-TON-1 and the expanded discussion in the general code quality recommendations appendix).

19. Missing base copy constructor invocation in derived copy constructor

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-19
Target: `validator/impl/shard.cpp`	

Description

The copy constructor for `ShardStateQ` does not invoke its parent class's copy constructor (see Figure 19.1). This issue can lead to unexpected behavior. `ShardStateQ` ultimately inherits from `CntObject`, which is part of the implemented reference counting mechanism. In this case, the `CntObject` copy constructor just invokes the default constructor, and the implementation works as expected. However, should there be any custom behavior in the `CntObject` copy constructor, reference counting could potentially fail, leading to memory leaks or use-after-free situations.

```
39    ShardStateQ::ShardStateQ(const ShardStateQ& other)
40       : blkid(other.blkid)
41       , rhash(other.rhash)
42       , data(other.data.is_null() ? td::BufferSlice{} : other.data.clone())
43       , bocs_(other.bocs_)
44       , root(other.root)
45       , lt(other.lt)
46       , utime(other.utime)
47       , before_split_(other.before_split_)
48       , fake_split_(other.fake_split_)
49       , fake_merge_(other.fake_merge_) {
50    }
```

Figure 19.1: Base class copy constructor not invoked (validator/impl/shard.cpp#L39-L50)

This finding is informational because the implementation currently works. Should there be any custom behavior in any of the base classes (e.g., as the result of a future refactor, optimization, or feature addition) this could become a serious issue, as it is involved in memory management.

The code example in Figures 19.2 and 19.3 illustrates the issue.

Additional instances of the same issue occur at the following three locations:

- crypto/block/block-db.cpp#L822-L829
- validator/impl/block.cpp#L27
- validator/impl/top-shard-descr.hpp#L36

```
#include <iostream>

class Base {
public:
  int member;
  Base() : member(1) {}
  Base(Base const& c) : member(c.member) {}
};

class Between : public Base {

};

class Derived : public Between  {
  int value;
  public:
  Derived() : value(1) {}

  Derived(Derived const& c) : value(c.value) {}
};

int main(int argc, char *argv[]) {
  Base b0;
  b0.member = 11;
  Base b1(b0);
  std::cout << "b1.member: " << b1.member << " equal after copy: " << (b1.member ==
b0.member) << std::endl;

  Derived d0;
  d0.member = 128;
  Derived d1(d0);
  std::cout << "d1.member: " << d1.member<< " equal after copy: " << (d1.member ==
d0.member) << std::endl;

  return 0;
}
```

Figure 19.2: Code example illustrating not invoking base class copy constructor from derived class copy constructor

```
#include <iostream>
$ g++ -O3 -std=c++17 parent_copy_ctor.cpp && ./a.out
b1.member: 11 equal after copy: 1
d1.member: 1 equal after copy: 0
```

Figure 19.3: Executing the program from Figure 19.2

Exploit Scenario

A change to one of the base classes is made and introduces a use-after-free bug or a memory leak, which an attacker exploits to get arbitrary code execution or cause the node to crash.

Recommendations

Short term, invoke the base class constructor to ensure consistent behavior.

Long term, implement static code analysis to detect when a copy constructor in a derived class does not invoke its base class copy constructor.

20. Unbounded storage of received Catchain blocks

Severity: **Informational**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-TON-20
Target: `catchain/catchain-receiver.cpp`	

Description
A feature of the Catchain protocol is that received blocks are queued pending arrival of dependent blocks. The number of blocks that are queued is unbounded unless a configuration setting is changed.

When the configuration parameter `catchain_max_blocks_coeff` is configured, nodes will reject blocks whose height is too high, limiting the risk of out-of-memory conditions.

The severity of this finding is informational because this is a documented vulnerability with an implemented mitigation. However, the default Catchain settings do not enable the mitigation.

Exploit Scenario
A malicious validator sends blocks to other validators that have dependencies that cannot be verified, causing high memory load on other nodes and eventually exhausting memory.

Recommendations
Short term, warn the user whenever the `catchain_max_blocks_coeff` parameter is set to zero.

Long term, prefer settings that are secure by default and consider changing the default value of `catchain_max_blocks_coeff` to the recommended value.

21. Getting account state can crash when building a state root proof

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-21
Target: `validator/impl/liteserver.cpp`	

Description

The function call to `get_hash(0)` on line 993 of `liteserver.cpp`, shown in Figure 21.1, returns a temporary object of type `vm::cell::Hash`. When invoking `bits()` on that object, a pointer referencing a member of the temporary object is returned and assigned to `upd_hash`. Once evaluation of that statement is complete, the temporary object goes out of scope and is destroyed. This leaves a dangling pointer in `upd_hash`.

The expression on line 994 (Figure 21.1) contains the same issue.

```
993    auto upd_hash = upd_cs.prefetch_ref(1)->get_hash(0).bits();
994    auto state_hash = state_root->get_hash().bits();
995    if (upd_hash.compare(state_hash, 256)) {
```

Figure 21.1: upd_hash points to out-of-scope stack memory after line 993 (validator/impl/liteserver.cpp#L993-L995)

Exploit Scenario

The TON codebase is built using a compiler that reuses the memory pointed to by `upd_hash` immediately after line 993. On line 995, the dereference causes a crash, causing the network to break consensus.

Recommendations

Short term, assign the return value of `get_hash()` to a variable to keep it alive for as long as there are pointers referring to the memory in it.

Long term, consider code patterns that prevent dangling pointers and implement dynamic code analysis to detect stack use after scope.

22. Misaligned object allocation and interaction

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-22
Target: `catchain/catchain-receiver.cpp`	

Description

A custom allocation scheme is implemented in `validator-session/validator-session-description.cpp`. Memory allocation eventually reaches the `ValidatorSessionDescriptionImpl::alloc` function (Figure 22.1). The `align` parameter controls alignment, but it is not used in the function body. The effect is that object allocation is executed as if aligned on 1-byte boundaries. However, objects with higher requirements on allocation (e.g., 8-byte alignment) receive memory from this function. This causes misaligned object construction. Interaction with these misaligned objects happens in several locations, triggering multiple instances of undefined behavior.

```
 164    void *ValidatorSessionDescriptionImpl::alloc(size_t size, size_t align, bool
temp) {
 165     if (temp) {
 166       auto s = pdata_temp_ptr_;
 167       pdata_temp_ptr_ += size;
 168       CHECK(s + size <= pdata_temp_size_);
 169       return static_cast<void *>(pdata_temp_ + s);
 170     } else {
 171       while (true) {
 172         auto s = pdata_perm_ptr_;
 173         pdata_perm_ptr_ += size;
 174
 175         if (pdata_perm_ptr_ <= pdata_perm_.size() * pdata_perm_size_) {
 176           return static_cast<void *>(pdata_perm_[s / pdata_perm_size_] + (s %
pdata_perm_size_));
 177         }
 178
 179         pdata_perm_.push_back(new td::uint8[pdata_perm_size_]);
 180       }
 181     }
 182    }
```

Figure 22.1: Root cause for non-aligned object construction (validator-session/validator-session-description.cpp#L164-L182)

Exploit Scenario
The TON node is built using a compiler that detects this undefined behavior and optimizes it out. This leads to a crash or incorrect computation during validation, causing the network to lose consensus.

Recommendations
Short term, implement aligned allocation by honoring the `align` parameter.

Long term, implement dynamic analysis to detect misaligned object allocation and interaction. Consider the use of the c++ keyword `alignof` to get alignment requirements.

23. Use of DowncastHelper leads to invalid downcast of incorrect type

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-23
Target: `tl/tl/tl_json.h`	

Description

The use of `downcast_call` from the auto-generated `ton_api.hpp` in combination with `DowncastHelper` from `tl/tl/tl_json.h#L240-L253` invokes undefined behavior due to an incorrect downcast.

One example is parsing of public keys from JSON, invoking the code in Figure 23.1. In this particular case, the helper will be of type `DowncastHelper<PublicKey>`, which inherits from `PublicKey` (see Figure 23.2). Because of the inheritance relation, the upcast to `PublicKey&` on line 279 (see Figure 23.1) is correct. However, in the specialization of `downcast_call` for `PublicKey` (Figure 23.3), the `obj` (who is the `helper` from Figure 23.1) is downcast into `pub_ed25519`, which also inherits from `PublicKey`. The issue is that `pub_ed25519` does *not* inherit from `DowncastHelper<PublicKey>`, which is the actual type of `obj`. Performing the downcast to `pub_ed25519` is undefined behavior.

```
277    DowncastHelper<T> helper(constructor);
278    Status status;
279    bool ok = downcast_call(static_cast<T &>(helper), [&](auto &dummy) {
280     auto result = ton::create_tl_object<std::decay_t<decltype(dummy)>>();
281     status = from_json(*result, object);
282     to = std::move(result);
283    });
```

Figure 23.1: Calling code, leading up to incorrect downcast in `downcast_call`. (`tl/tl/tl_json.h#L277-L283`)

```
240    template <class T>
241    class DowncastHelper : public T {
242    public:
243     explicit DowncastHelper(int32 constructor) : constructor_(constructor) {
244     }
245     int32 get_id() const override {
246       return constructor_;
247     }
248     void store(TlStorerToString &s, const char *field_name) const override {
249     }
```

```
250
251    private:
252     int32 constructor_{0};
253    };
```

Figure 23.2: Helper type used in JSON handling for constructing objects based on constructor ID (tl/tl/tl_json.h#L240-L253)

```
1330    template <class T>
1331    bool downcast_call(PublicKey &obj, const T &func) {
1332     switch (obj.get_id()) {
1333       case pub_unenc::ID:
1334         func(static_cast<pub_unenc &>(obj));
1335         return true;
1336       case pub_ed25519::ID:
1337         func(static_cast<pub_ed25519 &>(obj));
1338         return true;
1339       case pub_aes::ID:
1340         func(static_cast<pub_aes &>(obj));
1341         return true;
1342       case pub_overlay::ID:
1343         func(static_cast<pub_overlay &>(obj));
1344         return true;
1345       default:
1346         return false;
1347     }
1348    }
```

Figure 23.3: Specialization of `downcast_call` for `PublicKey`, invoking undefined behavior due to an incorrect downcast. (`tl/generate/auto/tl/ton_api.hpp:1330-1348`)

Exploit Scenario

The TON node is built using a compiler that detects this undefined behavior and optimizes it out. This leads to a crash or incorrect computation during validation, causing the network to lose consensus.

Recommendations

Short term, replace the use of `DowncastHelper` and `downcast_call` with an implementation that does not cause invalid downcasts. Figure 23.4 provides one example (pseudocode) of how this can be done.

Long term, implement dynamic code analysis to automatically detect incorrect downcasts during testing.

```
template <class T>
bool downcast_construct(PublicKey &obj, const T &func) {
 switch (obj.get_id()) {
   case pub_ed25519::ID:
     func(ton::create_tl_object<pub_ed25519>()>);
     return true;
   case pub_aes::ID:
     func(ton::create_tl_object<pub_aes>()>);
     return true;
   default:
     return false;
 }
}

DowncastHelper<T> helper(constructor);
Status status;
bool ok = downcast_construct(static_cast<T &>(helper), [&](auto result) {
 status = from_json(*result, object);
 to = std::move(result);
});
```

Figure 23.4: Example of how to construct the objects from constructor ID without causing invalid downcasts

24. Clock drift can break consensus

Severity: **Informational**	Difficulty: **Low**
Type: Timing	Finding ID: TOB-TON-24
Target: `validator-engine`	

Description
The TON Catchain block consensus protocol relies on nodes' local times in order to calculate the current round and attempt. The protocol assumes that all nodes have a globally synchronized clock; otherwise, nodes' calculated rounds and attempts may be incorrect. Our experiments have revealed that at most twenty seconds of clock drift is sufficient to prevent a node from participating in consensus. If one third or more of the nodes do not agree on the current time (and, thereby, the current round and attempt), then the consensus protocol will never quiesce. Over the past month, TON averaged 215 validators, so clock drift in at least 72 would be sufficient to deny service to the network.

Exploit Scenario
Over one third of the nodes' clocks do not agree on the current time. The consensus protocol never progresses, and the TON network does not accept any new blocks.

Recommendations
Short term, document the importance of synchronizing the nodes' clocks.

Long term, consider revising the consensus protocol such that it does not rely on nodes having synchronized clocks. Alternatively, consider switching to a well studied, partially synchronous BFT protocol like HotStuff. This would not increase the one third of nodes necessary to disrupt the system; however, it would prevent an unintentional denial of service in the presence of benign clock drift.

25. Shard records can be instantiated with uninitialized member variables

Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-25
Target: `crypto/block/block-parse.h`	

Description
The default constructor for the `ShardIdent::Record` class does not initialize its `workchain_id` and `shard_prefix` member variables.

```
981    struct ShardIdent::Record {
982     int shard_pfx_bits;
983     int workchain_id;
984     unsigned long long shard_prefix;
985     Record() : shard_pfx_bits(-1) {
986     }
```

Figure 25.1: The default constructor only initializes the `shard_pfx_bits` member. (`crypto/block/block-parse.h#981–986`)

The severity of this finding is undetermined because we were unable to confirm whether the uninitialized members are ever read from an object initialized this way.

Exploit Scenario
A shard record is initialized with the default constructor, and its prefix is read before being initialized, causing a crash.

Recommendations
Short term, initialize all of the member variables from the `ShardIdent::Record` default constructor.

Long term, integrate static analyses into TON's CI lifecycle that would catch these sorts of errors.

26. Signatures of block antecessors are not validated

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TON-26
Target: `validator/impl/validate-query.cpp`	

Description
Antecessor block signature verification has yet to be implemented. As a result, nothing is executed if there is at least one signature (the first case on line 5422 of Figure 26.1). The rejection on line 5424 is inaccessible because of the "`&& false`" on line 5423.

```
 5420    if (id_.seqno() > 1) {
 5421     if (prev_signatures_.not_null()) {
 5422       // TODO: check signatures here
 5423     } else if (!is_fake_ && false) {  // FIXME: remove "&& false" when
collator serializes signatures
 5424       return reject_query("block contains an empty signature set for the
previous block");
 5425     }
 5426    }
```

Figure 26.1: Signatures on the previous block are not validated.
(validator/impl/validate-query.cpp#5420–5426)

This code path is exercised when validating master chain block queries.

The severity of this finding is undetermined because it is unclear if a block with invalid antecessor signature(s) could pose a security risk.

Exploit Scenario
A malicious user or validator crafts a block whose antecessor signatures are incorrect. Other validators accept this block, even though it should be rejected. This leads to a fork.

Recommendations
Short term, validate antecessor signatures.

Long term, add unit tests that exercise this edge case.

27. TLB reference validation can be bypassed

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-TON-27
Target: `crypto/tl/tlblib.cpp`	

Description

The `TLB::validate_ref_internal` implementation blindly decrements an argument before testing whether it is negative. If the value is `INT_MIN`, then the ensuing integer underflow will cause the test to return an erroneous value.

```
 127    bool TLB::validate_ref_internal(int* ops, Ref<vm::Cell> cell_ref, bool weak)
const {
 128     if (ops && --*ops < 0) {
 129       return false;
 130     }
```

Figure 27.1: Integer underflow can occur on line 128 if the `ops` argument has value `INT_MIN`. (`crypto/tl/tlblib.cpp#127–130`)

The severity and difficulty of this finding is undetermined because we were unable to confirm whether the ops argument is user-controllable to the extent that it could be set to `INT_MIN`.

Exploit Scenario

A malicious user crafts a message that causes the `ops` argument to underflow, causing the validation to pass when it should have failed.

Recommendations

Short term, confirm if the `ops` pointee is user-controlled and implement a check for integer underflow.

Long term, ensure test cases explore the boundaries of validation.

28. The TON client's get shards request can fail

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-28
Target: `tl/tl/tl_json.h`	

Description

The lambda used to handle liteserver query responses for obtaining all shards' information has several return statements commented out. For example, a deserialization error will cause the code on line 4216 of Figure 28.1 to fall through and continue as if the error did not occur. If either of the checks on lines 4211 or 4220 succeeds, the function will flow off the end without returning a value, which constitutes undefined behavior.

```
 4208    promise.wrap([](lite_api_ptr<ton::lite_api::liteServer_allShardsInfo>&&
all_shards_info) {
 4209      td::BufferSlice proof = std::move((*all_shards_info).proof_);
 4210      td::BufferSlice data = std::move((*all_shards_info).data_);
 4211      if (data.empty()) {
 4212        //return td::Status::Error("shard configuration is empty");
 4213      } else {
 4214        auto R = vm::std_boc_deserialize(data.clone());
 4215        if (R.is_error()) {
 4216          //return td::Status::Error("cannot deserialize shard configuration");
 4217        }
 4218        auto root = R.move_as_ok();
 4219        block::ShardConfig sh_conf;
 4220        if (!sh_conf.unpack(vm::load_cell_slice_ref(root))) {
 4221          //return td::Status::Error("cannot extract shard block list from
shard configuration");
 4222        } else {
 4223          auto ids = sh_conf.get_shard_hash_ids(true);
 4224          tonlib_api::blocks_shards shards;
 4225          for (auto id : ids) {
 4226            auto ref = sh_conf.get_shard_hash(ton::ShardIdFull(id));
 4227            if (ref.not_null()) {
 4228              shards.shards_.push_back(to_tonlib_api(ref->top_block_id()));
 4229            }
 4230          }
 4231         return
tonlib_api::make_object<tonlib_api::blocks_shards>(std::move(shards));
 4232        }
 4233      }
 4234    }));
```

Figure 28.1: (tonlib/tonlib/TonlibClient.cpp#4208–4234)

Exploit Scenario

A TonLibClient receiving shards of unexpected format triggers undefined behavior, leading to a crash or incorrect computation.

Recommendations

Short term, restore the disabled return statements to prevent invocation of undefined behavior.

Long term, implement static code analysis to detect functions that are missing return statements.

29. Bigint and cell tests can silently fail due to undefined behavior

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-29
Target: `crypto/test/test-bigint.cpp /modbigint.cpp /test-cells.cpp`	

Description

The `test-bigint.cpp` test exhibits undefined behavior by executing a left shift of a negative value on lines 189 (Figure 29.1) and 218 (Figure 29.2).

```
187    for (int i = 63; i >= 0; --i) {
188     if (r < 8) {
189       acc += (a << r);
190       r = 1024;
191     }
192     r -= 8;
193     bin[i] = (unsigned char)(acc & 0xff);
194     acc >>= 8;
195    }
```

Figure 29.1: On line 189, the variable a has value -32 for at least one iteration of the test. (crypto/test/test-bigint.cpp#187–195)

```
217    void bin_add_small(unsigned char bin[64], long long val, int shift = 0) {
218     val <<= shift & 7;
219     for (int i = 63 - (shift >> 3); i >= 0 && val; --i) {
220       val += bin[i];
221       bin[i] = (unsigned char)val;
222       val >>= 8;
223     }
224    }
```

Figure 29.2: On line 218, `val` has value -1 for at least one iteration of the test. (crypto/test/test-bigint.cpp#217–224)

The `modbigint.cpp` test exhibits similar undefined behavior on line 906 (Figure 29.3) and also exhibits signed integer overflow on line 294 (Figure 29.4).

```
904    for (; i < size; i++) {
905     pow += 8;
906     acc = (acc << 8) + arr[i];
907     if (pow >= 56) {
908       lshift_add(pow, acc);
```

```
909         acc = pow = 0;
910       }
911     }
```

Figure 29.3: On line 906, `acc` has value -1 for at least one iteration of the test. (`crypto/test/modbigint.cpp#904–911`)

```
291     explicit operator long long() const {
292       long long acc = 0.;
293       for (int i = N - 1; i >= 0; --i) {
294         acc = acc * mod[i] + a[i];
295       }
296       return acc;
297     }
```

Figure 29.4: On line 294, `acc` has value 420121321411714226 and `mod[i]` has value 999999937 for at least one iteration of the test, causing signed integer overflow. (`crypto/test/modbigint.cpp#291–297`)

Finally, the `test-cells.cpp` test also exhibits signed integer overflow on line 553 (Figure 29.5).

```
551     for (auto& c : r) {
552       c = (k & 0x80) ? (unsigned char)(k >> 8) : 0;
553       k = 69069 * k + 1;
554     }
```

Figure 29.5: On line 553, `k` has value 69070 on at least one iteration of the test, causing signed integer overflow in the multiplication. (`crypto/test/test-cells.cpp#551–554`)

Exploit Scenario

The compiler detects the undefined behavior and elides the code during optimization, causing the tests to erroneously pass when they should have failed, hiding a latent bug.

Recommendations

Short term, add data validation to ensure that the undefined behavior does not occur.

Long term, regularly compile and run all tests with the LLVM undefined behavior sanitizer enabled (UBSan, see Appendix D), preferably in TON's CI pipeline.

30. Multiplication of a constant can lead to a misaligned stack

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TON-30
Target: `crypto/func/builtins.cpp`	

Description

The FunC code in Figure 30.1 contains a function that should always return a single integer, valued zero.

```
int test(int x) {
  return (x * 0) * 0;
}
```

Figure 30.1: The `test` function always returns zero, regardless of the value of x.

When compiled, the `test` function results in the Fift code in Figure 30.2.

```
DECLPROC test
test PROC:<{
  0 MULCONST
  0 PUSHINT
}>
```

Figure 30.2: The resulting Fift code when the `test` function from Figure 30.1 is compiled.

In FunC's calling convention, the function is responsible for cleaning up the stack before returning. In this case, the final `0 PUSHINT` instruction causes an extraneous value to remain on the stack after the function returns, resulting in a stack misalignment.

Exploit Scenario

Consider the FunC code in Figure 30.3.

```
int test(int x) {
    return 42 | (x * 0 * 0);
}

() main() {
    var test_result = test(0);
    throw_unless(100, test_result == 42);
}
```

Figure 30.3: A real-world example of how stack misalignment can lead to a bug

This code should never throw an exception since the result of the `test` function will always be 42 regardless of the value of `x`. However, the stack after the call to `test(0)` will be 42 at the bottom and zero at the top (thanks to the erroneous `PUSHINT`, as demonstrated in Figure 30.2). Therefore, the value of `test_result` will be zero due to the stack misalignment, and an exception will be thrown.

The code is deployed on mainnet and is interpreted differently than the programmer intended.

Recommendations

Short term, fix the multiplication stack misalignment. We suspect that it is due to a variable not being marked as unused during a multiply-by-zero optimization.

Long term, increase unit test and fuzzing coverage to explore these edge cases.

31. FunC codegen invokes undefined behavior

Severity: **Medium**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-31
Target: `crypto/func/codegen.cpp`	

Description

When generating the next code step, the operator's `next` member variable is always dereferenced even though it can sometimes be `nullptr`. This will happen on line 279 of Figure 31.1. This undefined behavior can lead to a crash.

```
275    bool Op::generate_code_step(Stack& stack) {
276     stack.opt_show();
277     stack.drop_vars_except(var_info);
278     stack.opt_show();
279     const auto& next_var_info = next->var_info;
280     bool inline_func = stack.mode & Stack::_InlineFunc;
```

Figure 31.1: The next variable can be `nullptr` when dereferenced on line 279. (crypto/func/codegen.cpp#275–280)

Exploit Scenario

This undefined behavior is actually exhibited when compiling the elector code contract:

```
$ crypto/func -PS -o /tmp/dst.fif smartcont/stdlib.fc
  smartcont/elector-code.fc
```

However, the compilation completes without an error. The undefined behavior will cause a differential between compilers, and potentially introduce errors into the resulting Fift code. This can be confirmed by adding an assertion between lines 278 and 279 that ensures

`next != nullptr`.

Recommendations

Short term, ensure that `next` cannot be `nullptr` before attempting member access. The `next_var_info` variable is not used in every code path within the function, so also consider calculating it only when necessary.

Long term, implement dynamic code analysis to detect member accesses within null pointers.

32. Constant operations on NaN can cause the FunC compiler to crash

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-32
Target: `crypto/common/refcnt.hpp`	

Description
Constant operations on NaN can cause the FunC compiler to crash in different ways. For example, the code in Figure 32.1 will cause the compiler to abort due to a failed reference counting assertion.

```
int eval(int x) {
    return x / 0 + 0 * x + x;
}
```

Figure 32.1: An example FunC program that will cause the compiler to crash

Both the division by zero and the multiplication by zero are necessary to produce the crash. Swapping the first two addends will also produce the same crash; however, swapping the latter two addends will enable compilation without a crash. This suggests that the bug may be attributable to constant operations on NaN, since there is an existing codegen optimization that will rewrite `x / 0` as `NaN` (even if peephole optimizations are disabled with `-O0`).

If the last addend—the standalone `x` term—is changed to a constant, then the compiler will abort with a different crash:

```
libc++abi: terminating with uncaught exception of type
td::CntObject::WriteError
```

Exploit Scenario
The FunC compiler crashes, failing to produce an intelligible error, for FunC code with valid syntax.

Recommendations
Short term, fix these bugs to ensure that the compiler does not crash on valid FunC input.

Long term, add unit tests to cover this edge case.

33. Undefined variables in FunC are treated as undefined functions and do not cause a compiler error

Severity: **Medium**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-TON-33
Target: `crypto/func/codegen.cpp`	

Description
Undefined variables in FunC code are treated as undefined function symbols. An error is printed, but the compiler still emits Fift code as if the variables were functions and exits with return code zero.

Exploit Scenario
Consider the code in Figure 33.1:

```
1    int contains_typo(int a) {
2     return a*2;
3    }
```

Figure 33.1: A FunC function containing a typo on line 2

Unlike most programming languages, FunC requires whitespace around operators. However, the programmer made the honest mistake of forgetting to add whitespace around the multiplication operator on line 2. The FunC compiler will print a warning, yet it will still emit Fift code that erroneously treats the variable as an undefined function symbol. Since the compiler exits with return code zero, the programmer is likely to miss this error, particularly if there is a significant number of log messages thereafter.

Recommendations
Short term, treat undefined symbols in contexts where they are used as a variable (rather than a call) as unrecoverable compiler errors. Exit the compiler with a non-zero exit code.

Long term, add unit tests to cover this edge case.

34. Calls to implicitly impure functions without a return value are always optimized out without an error

Severity: **Medium**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-TON-34
Target: FunC	

Description

Calls to `impure_function()` without a return value will always be optimized out without producing a warning or error.

Consider the FunC code in Figure 34.1.

```
global int G;

() impure_function() {
    G = 5;
}

() main() {
    impure_function();
}
```

Figure 34.1: The call to `impure_function` from `main` will be elided since it does not have a return value and is not marked as `impure`.

Although the `impure_function()` is implicitly impure because it modifies a global as a side effect, it was not explicitly marked as `impure`. Therefore, the compiler is permitted to optimize out calls to it where the return value is not used. However, since `impure_function()` has no return value, it will *always* be optimized out, even if optimizations are disabled with `-O0`. This will produce neither a compiler warning nor a compiler error.

Any function with no return value that is *not* explicitly marked as `impure` is almost certainly a programming error and should be caught by the compiler.

Exploit Scenario

The programmer forgot to explicitly mark an implicitly impure function with the `impure` specifier. Calls to the function are optimized out without warning, despite the fact that the calls would have had side effects.

Recommendations

Short term, issue a compiler warning for every function with neither return values nor the `impure` specifier.

Long term, add unit tests to cover this edge case, and consider elevating the compiler warning to an irrecoverable compiler error.

35. Calls to implicitly impure functions with unused return values are always optimized out without an error

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-TON-35
Target: FunC	

Description

Similarly to TOB-TON-34, calls to implicitly impure functions with *unused* return values are always optimized out without producing a compiler warning or error.

Consider the FunC code in Figure 35.1. It illustrates a potential input validation scenario.

```
;; Ensure arg is valid (>123)
int validate_arg(int arg) {
 throw_unless(111, arg > 123);
 return arg;
}

int main(int val) {
 int v = validate_arg(val);

 ;; Note that v is unused
 return val * 2;
}
```

Figure 35.1: Program validation logic is missing an impure specifier.

The generated code for Figure 35.1 can be found in Figure 35.2.

```
DECLPROC validate_arg
DECLPROC main
validate_arg PROC:<{
 DUP
 123 GTINT
 111 THROWIFNOT
}>
main PROC:<{
 1 LSHIFT#
}>
```

Figure 35.2: Resulting Fift code when compiling the FunC code from Figure 35.1.

Note that the compiler silently elided the call to `validate_arg` from `main`. This is because the `validate_arg()` function was not explicitly marked as `impure`, despite the fact that it is implicitly impure due to its side effect of throwing. The compiler is permitted to optimize out calls to it when the return value of `validate_arg()` is assigned to variable v that is unused (see Figure 35.2). This will produce neither a compiler warning nor a compiler error.

Any function that is *not* explicitly marked impure having an unused return value is likely a programming error and should be caught by the compiler.

Despite being very similar to TOB-TON-34, this finding is informational because it is technically documented behavior. However, we assume that this behavior is unexpected for most programmers learning FunC, since most other programming languages do not involve this sort of silent elision. Therefore, we strongly recommend TON consider our recommendations to this finding.

Exploit Scenario

The programmer has forgotten to explicitly mark an implicitly impure function with the `impure` specifier. Calls to the function are optimized out without warning, even though the calls would have had side effects. In the above case, validation would be skipped, silently enabling attacks.

Recommendations

Short term, issue a compiler warning for every call to a function that has a return value where the return value is unused, regardless of purity. For functions not marked as `impure`, consider elevating this to an irrecoverable compiler error.

Long term, add unit tests to cover this edge case, and consider elevating the compiler warning to a compiler error.

36. Comparison to NaN results in the other comparand

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-TON-36
Target: `crypto/vm/arithops.cpp`	

Description

In the `exec_cmp` function (see Figure 36.1), if either x or y is not valid then x is chosen as the result of the operation (possibly throwing if the operation is not `quiet` and x is not valid). However, for the case that y is not valid, the result will always be x and, unless x is zero, will evaluate to a `true` expression. Therefore, if x is not zero and y is not valid (e.g., if y is NaN), then x and y will always be considered equal, regardless of their values.

```
851    if (!x->is_valid() || !y->is_valid()) {
852     stack.push_int_quiet(std::move(x), quiet);
853    } else {
```

Figure 36.1: Vulnerability that causes comparison to NaN to result in the other comparand. (`crypto/vm/arithops.cpp#851–853`)

All comparison operators—not just equality (==), but also <, >=, etc.—use the `exec_cmp` function and are therefore vulnerable to this behavior.

The FunC code in Figure 36.2 contains a procedure that compares the return value of the result of the `qufits` function to 64. The `qufits` function silently returns NaN in this case.

```
1    int qufits(int x, int bits) impure asm "QUFITSX";
2
3    () evaluate(int l, int r) impure {
4           throw_unless(345, l == r);
5    }
6    int main() {
7           evaluate(65, qufits(1000000000000000, 1));
8           return 0;
9    }
```

Figure 36.2: The `test` function always returns zero, regardless of the value of x.

When compiled, the `test` function results in the Fift code shown below.

```
1    "Asm.fif" include
2    // automatically generated from `../crypto/func/opttest/TOB-TON-36.fc`
```

```
3    PROGRAM{
4     DECLPROC evaluate
5     DECLPROC main
6     evaluate PROC:<{
7       EQUAL
8       345 THROWIFNOT
9     }>
10    main PROC:<{
11      65 PUSHINT
12      1000000000000000 PUSHINT
13      1 PUSHINT
14      QUFITSX
15      evaluate CALLDICT
16      0 PUSHINT
17    }>
18   }END>
```

Figure 36.3: The resulting Fift code when the function from Figure 36.1 is compiled.

One would expect this code to always throw, since `65 != NaN`. However, the code will not throw when it is executed because of the logic bug in the `exec_cmp` function.

Exploit Scenario

Smart contract code is deployed on mainnet and an attacker passes parameters to a function, resulting in a comparison to `NaN`. The comparison succeeds even though it should have failed, allowing the attacker to bypass data validation controls.

Recommendations

Short term, make operator behavior with `NaN` operands consistent, regardless of argument order. Consider making comparison operators behave similarly to how other languages handle `NaN` comparisons. For example, in most languages (including Python and JavaScript), `NaN` does not equal any other value, including itself.

Long term, increase unit test and fuzzing coverage to explore these edge cases. Consider throwing an exception in the event that a `NaN` is an argument to a non-silent comparison operator (other than `ISNAN`).

37. FunC fails to reject out-of-range constants

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TON-37
Target: `func`	

Description

The FunC code in Figure 37.1 throws unless two constants are equal. The constants are outside of the valid range of -2^{256} to 2^{256}.

```
1    () evaluate(int l, int r) impure {
2            throw_unless(345, l == r);
3    }
4    int main() {
5
evaluate(-165040263640452188282900436258425884295597337378342696517441825933514918852577,
-165040263640452188282900436258425884295597337378342696517441825933514918852577);
6            return 0;
7    }
```

Figure 37.1: FunC code with constants out of valid range

The successfully compiled output is shown below.

```
  1    "Asm.fif" include
  2    // automatically generated from `../crypto/func/opttest/TOB-TON-37.fc`
  3    PROGRAM{
  4     DECLPROC evaluate
  5     DECLPROC main
  6     evaluate PROC:<{
  7       EQUAL
  8       345 THROWIFNOT
  9     }>
 10     main PROC:<{
 11
-1650402636404521882829004362584258842955973373783426965174418259335149188525 77
PUSHINT
 12       DUP
 13       evaluate CALLDICT
 14       0 PUSHINT
 15     }>
 16    }END>c
```

Figure 37.2: Generated Fift code with invalid constants

When attempting execution using `fift`, the code produces a runtime error (Figure 37.3).

```
 1    $ ./crypto/fift -I ../crypto/fift/lib/ ./tob-ton-36.fif
 2    [ 1][t 0][2022-10-06 07:28:28.520341537][Fift.cpp:67]   top: <text interpreter
continuation>
 3    [ 1][t 0][2022-10-06 07:28:28.520520772][fift-main.cpp:204]     Error
interpreting file `./tob-ton-36.fif`: tob-ton-36.fif:11:
-1650402636404521882829004362584258842955973373783426965174418259335149188525 77:-?
```

Figure 37.3: Attempted execution of Fift code from Figure 37.2

Exploit Scenario

The FunC compiler fails to validate the range of integral constants, resulting in incorrect code generation or code that will always cause an exception at runtime.

Recommendations

Short term, ensure that `func` fails with an error message for literal integer constants that are out of range.

Long term, increase unit test and fuzzing coverage to explore these edge cases.

38. Inconsistent runtime behavior for operations resulting in NaN

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TON-38
Target: `func`	

Description

Consider the FunC code in Figure 38.1. The procedures `f1` and `f2` perform the same operation and differ only by how the denominator of the division is defined: Whereas `f1` immediately divides by zero, `f2` accepts the denominator as an argument.

```
1    int f1(int arg) {
2     var v = arg / 0;
3     if (arg == 3) {
4       return v;
5     } else {
6       return 99;
7     }
8    }
9
10   int f2(int arg, int denom) {
11    var v = arg / denom;
12    if (arg == 3) {
13      return v;
14    } else {
15      return 99;
16    }
17   }
18
19   int main() {
20    return f1(2);
21    ;; return f2(2, 0);
22   }
```

Figure 38.1: Data dependent evaluation that could result in operations on NaN values

The successfully compiled output is shown below:

```
1    DECLPROC f1
2    DECLPROC f2
3    DECLPROC main
4    f1 PROC:<{
5     PUSHNAN
6     SWAP
```

```
 7      3 EQINT
 8      IFJMP:<{
 9      }>
10      DROP
11      99 PUSHINT
12     }>
13     f2 PROC:<{
14      s1 s(-1) PUXC
15      DIV
16      SWAP
17      3 EQINT
18      IFJMP:<{
19      }>
20      DROP
21      99 PUSHINT
22     }>
23     main PROC:<{
24      2 PUSHINT
25      f1 CALLDICT
26     }>
```

Figure 38.2: Resulting fift-code from func-code in Figure 38.1

When `f1` is invoked, as occurs in `main`, the division-by-zero on line 2 of Figure 38.1 pushes a `NaN` to the stack on line 5 of Figure 38.2, and the function will always successfully return 99 (unless `arg` equals 3). However, if a semantically equivalent call to `f2` is made with arguments `arg=2` and `denom=0`, the division on line 15 of Figure 38.2 will always throw an exception, regardless of the value of `arg`. This is because division-by-zero *at runtime* always throws an exception.

The following operators have been confirmed to have the same behavior for operations involving division-by-zero: `~/`, `^/`, `/=`, `~/=`, `^/=`, `%`, `%=`, `~%=`, and `^%=`. However, this behavior does not seem to apply to operator `/%`.

Bit shifts with out-of-range shift amounts (e.g., a left shift of 257 bits) result in similar behavior: If the number of bits is a constant, then the code compiles and runs correctly, but if the same value is used at runtime, then an exception is thrown.

Exploit Scenario

Semantically equivalent operations have different behavior, causing unexpected control flow on deployed smart contracts.

Recommendations

Short term, have `func` generate consistent code for every expression, regardless of whether it is calculated at runtime or compile time. If semantically equivalent code would throw an exception at runtime, then semantically equivalent compile-time-calculated expressions should at a minimum produce Fift code that would produce the same

exception (e.g., code 4, the overflow exception). Since this scenario can be detected at compile time, a compiler warning should also be generated.

Long term, increase unit test and fuzzing coverage to explore these edge cases. Consider elevating this warning to an irrecoverable compiler error.

---

39. Missing _Bit-marker for positive integer 1

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TON-39
Target: `crypto/func/abscode.cpp`	

Description

Consider the code in Figure 39.1: It records properties of the represented integer constant based on its sign (line 150). If the integer constant has sign zero (line 160), indicating that the value is zero, then a check for if the value is one is made (line 161). If that check is successful, the `_Bit` marker is added (line 162). Clearly, the check on line 162 will never be true, since it is impossible for the sign (s) to be zero *and* for the constant to have value 1 at the same time.

It appears that the conditional on lines 161–163 was intended to belong in the preceding branch (between lines 159 and 160), since that is the branch that would be taken if `*int_const == 1`.

```
150    int s = sgn(int_const);
151    if (s < -1) {
152     val |= _Nan | _NonZero;
153    } else if (s < 0) {
154     val |= _NonZero | _Neg | _Finite;
155     if (*int_const == -1) {
156       val |= _Bool;
157     }
158    } else if (s > 0) {
159     val |= _NonZero | _Pos | _Finite;
160    } else if (!s) {
161     if (*int_const == 1) {
162       val |= _Bit;
163     }
164     val |= _Zero | _Neg | _Pos | _Finite | _Bool | _Bit;
165    }
166    if (val & _Finite) {
167     val |= int_const->get_bit(0) ? _Odd : _Even;
168    }
```

Figure 39.1: Source code responsible for recording properties of an integer as part of FunC code generation (`crypto/func/abscode.cpp#150–168`)

Exploit Scenario

The FunC compiler is later changed to rely on the `_Bit` marker for positive integers with value one. The `_Bit` marker is not set, resulting in unexpected control flow.

Recommendations

Short term, consider moving the misplaced conditional into the preceding branch.

Long term, increase unit test and fuzzing coverage to explore these edge cases.

40. Method IDs can collide without warning

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TON-40
Target: `func`	

Description

The `func` compiler will emit Fift code for methods that have duplicate method IDs without a warning or an error. For example, the code in Figure 40.1 results in the Fift in Figure 40.2.

```
1   int f1(int v) method_id(0) {
2    return v + 1;
3   }
4
5   int f2(int v) method_id(0) {
6    return v + 1;
7   }
```

Figure 40.1: Two FunC methods with duplicate IDs

```
 1   "Asm.fif" include
 2   // automatically generated from `../crypto/func/opttest/TOB-TON-40.fc`
 3   PROGRAM{
 4    0 DECLMETHOD f1
 5    0 DECLMETHOD f2
 6    f1 PROC:<{
 7      INC
 8    }>
 9    f2 PROC:<{
10      INC
11    }>
12   }END>c
```

Figure 40.2: The resulting fift code from compiling the FunC code in Figure 40.1.

When interpreted, this Fift code throws a runtime error, depicted in Figure 40.3.

```
[ 1][t 0][2022-10-03 12:48:57.905150229][Fift.cpp:67]   top: abort
level 1: swap { <continuation 0x604000017890> } if **HERE** drop
level 2: [in @PROC:<{:] over @fail-ifdef **HERE** 2 { <continuation 0x60400006ad50>
} does null swap @doafter<{ 0 32 u,
level 3: <text interpreter continuation>
[ 1][t 0][2022-10-03 12:48:57.905294049][Fift.cpp:70]   PROC:<{:procedure already
defined
[ 1][t 0][2022-10-03 12:48:57.906304393][Fift.cpp:67]   top: tuck
level 1: [in @addop:] tuck **HERE** sbitrefs @ensurebitrefs swap s,
level 2: <text interpreter continuation>
[ 1][t 0][2022-10-03 12:48:57.906356454][Fift.cpp:70]   INC:stack underflow
[ 1][t 0][2022-10-03 12:48:57.906594639][Fift.cpp:67]   top: abort
level 1: <text interpreter continuation>
[ 1][t 0][2022-10-03 12:48:57.906634237][Fift.cpp:70]   }>:not in asm context
```

Figure 40.3: Output from interpreting the Fift code from Figure 40.2.

The "`PROC:<{:procedure already defined`" error emitted by Fift does not provide any context about *which* procedure was already defined; if the FunC code contains many methods, this would be a difficult problem to debug.

The method ID collision was caused by explicitly setting the IDs in the original FunC code of Figure 40.1. However, these collisions can naturally occur due to FunC's use of CRC16 checksums to automatically generate IDs from procedure names. The address space of CRC16 is very small; collisions will naturally occur. For example, over 87% of method ID values (57,322 out of a possible 65,535) have at least two dictionary words that, if used as method names, would produce the same method ID. Such colliding names include:

- "balanced" and "secret"
- "balance", "get_abis", "get_askc", and "avlo"
- "seqno", "kjob", and "pconf"
- "get_public_key", "last_eods", and "nrk"
- "create_init_state", "xgk", "create_ahtg", and "init_yshw"
- "withdrawer" and "likelihood"
- "liquidate" and "burial"

Many of these names are used extensively throughout TON's smart contracts with auto-generated, CRC16-based method IDs.

Appendix I discusses how we discovered these method ID collisions and includes code for generating additional collisions.

Exploit Scenario

A smart contract author exposes two procedures whose auto-generated method IDs naturally collide, as in Figure 40.4. The FunC compiler neither emits an error nor a warning,

producing the Fift code in Figure 40.5. When the contract is deployed, a runtime error occurs.

```
(int) balanced() method_id { return 0; }
(int) secret() method_id { return 1; }
```

Figure 40.4: Two methods with auto-generated CRC16-based method IDs

```
69469 DECLMETHOD balanced
69469 DECLMETHOD secret
balanced PROC:<{
 0 PUSHINT
}>
secret PROC:<{
 1 PUSHINT
}>
```

Figure 40.5: Both methods are assigned the same ID: 69469

Recommendations

Short term, emit an irrecoverable compiler error if any two procedures have the same method ID. Ensure that the runtime error emitted by Fift when it discovers duplicate procedures additionally includes the offending procedure name(s).

Long term, consider switching to a different method of auto-generating method IDs that is less likely to cause collisions (e.g., a cryptographically secure hashing function with a larger address space).

41. Single-line comments are honored within multi-line comments

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TON-41
Target: `func`	

Description

The FunC parser honors single-line comments even when they occur inside multi-line comments. This can lead to visually confusing code, as in Figure 41.1, where the single-line comment will gobble the closing multi-line comment delimiter at the end of line 3.

```
1    {-
2      this is a multi-line comment
3      ;; this is a single-line comment inside of a multi-line comment -}
4
5    This is still inside the multi-line comment!
```

Figure 41.1: A single-line comment inside of a multi-line comment, gobbling the -} delimiter

Exploit Scenario

A malicious contract author wants to obfuscate the behavior of their contract. This is very common in the Ethereum ecosystem, where malicious contract authors deploy contracts that appear to reward anonymous users by interacting with them, but instead are honeypots that steal the users' funds.

Consider the code in Figure 41.2. The author has made it appear as if a caller can withdraw a lot of funds from the contract on lines 4 through 6, when in fact those lines are commented out due to the single-line comment on line 2. The multi-line comment in fact ends on line 51. A user sees that they might stand to gain a lot of money by calling `withdraw()`, as long as their current balance is at least 10. So the victim ensures that they transfer at least 10 into the contract before calling `withdraw()`. In fact, the code will use the implementation of `withdraw_amount()` on line 102 that returns zero, trapping the user's deposit in the contract.

```
  1    {- ;; this is a withdraw/deposit contract
  2       ;; anyone can withdraw all of the funds -}
  3
  4    (int) withdrawable_amount() method_id {
  5       return 1000000;
  6    }
  7
  8    (int, int) more_complicated_code_to_distract() {
  ⋮
 51    }-}
 52
 53    () withdraw() impure {
 54       ;; code to require the sender to have a balance of at least ten
 55       ;; code to transfer withdrawable_amount() to the sender
 56    }
 57
  ⋮
102    (int) withdrawable_amount() method_id { return 0; }
```

Figure 41.2: A malicious honeypot contract that is visually obfuscated. Everything before line 52 is actually a comment

Recommendations

Short term, document this behavior.

Long term, consider ignoring single-line comments within multi-line comments.

42. Bitwise operators can cause the FunC compiler to crash

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-42
Target: `crypto/func/builtins.cpp`	

Description

Consider the FunC program in Figure 42.1.

```
1    int nonzero() {
2     return 2;
3    }
4
5    int main() {
6     throw_if(111, 2 == (5 | 0 * nonzero()));
7     return 0;
8    }
```

Figure 42.1: FunC code causing the func-compiler to abort on a CHECK-failure

During compilation of the program, the func compiler will abort with a CHECK-failure at `refcnt.hpp` line 295. This is due to a `VarDescr` that is both `_Const` and `_Int`, but does not have an `int_const` member assigned.

The code in Figure 42.2 is responsible for compiling the or-expression.

```
511    AsmOp compile_or(std::vector<VarDescr>& res, std::vector<VarDescr>& args) {
512     assert(res.size() == 1 && args.size() == 2);
513     VarDescr &r = res[0], &x = args[0], &y = args[1];
514     if (x.is_int_const() && y.is_int_const()) {
515       r.set_const(x.int_const | y.int_const);
516       x.unused();
517       y.unused();
518       return push_const(r.int_const);
519     }
520     r.val = emulate_or(x.val, y.val);
521     return exec_op("OR", 2);
522    }
```

Figure 42.2: Compiler function responsible for the |-operator (`crypto/func/builtins.cpp#511–522`)

During compilation of the or-expression on line 6 of Figure 42.1, the x operand defined on line 513 of Figure 42.2 will be a constant int (5), while the y operand will not be constant.

This causes the `emulate_or` function (Figure 42.3) to be invoked using the respective argument characteristics (encoded in `val`).

```
194    int emulate_or(int a, int b) {
195     if (b & VarDescr::_Zero) {
196       return a;
197     } else if (a & VarDescr::_Zero) {
198       return b;
199     }
200     int both = a & b, any = a | b;
201     int r = VarDescr::_Int;
202     if (any & VarDescr::_Nan) {
203       return r | VarDescr::_Nan;
204     }
205     r |= VarDescr::_Finite;
206     r |= any & VarDescr::_NonZero;
207     r |= any & VarDescr::_Odd;
208     r |= both & VarDescr::_Even;
209     return r;
210    }
```

Figure 42.3: Function responsible for transforming operand characteristics to the resulting value for an | operation (`crypto/func/builtins.cpp#194–210`)

In this particular case, `b` will be `_Zero` due to the multiplication with 0 in on line 6 of Figure 42.1. This returns the characteristics for `a`. Since `a` is both `_Const` and `_Int`, various locations in the code will automatically assume that it has a non-null `int_const` member. However, as is apparent from Figure 42.2, the `int_const` is never set. This will cause the func compiler to abort with `CHECK` and `WriteError`-exceptions when the `int_const` member is accessed.

Additional similar discrepancies have been observed for other operators, such as `emulate_and`. For that particular case, it turns out that the `ConstZero` characteristic does not actually include the `_Const` marker, which would otherwise trigger additional CHECKs.

Exploit Scenario

A user writes valid FunC code, such as that of Figure 42.1. The compiler unexpectedly crashes with no indication of the offending code.

Recommendations

Ensure that the `emulate_*`-functions are in sync with the `compile_*` functions with respect to short circuit and constant operand behavior. For the particular case of `compile_or`, additional checks should be performed to see if the `int_const` member should be assigned and arguments marked as unused based on `r.val`. Further, consider if the `ConstZero`, `ConstOne`, and `ConstTrue` characteristics should include `_Const`.

Long term, add test cases covering short circuiting behavior of FunC operators. Consider

adding automated code generation and fuzzing to the TON CI pipeline to explore edge cases automatically.

43. FunC compiler can produce undefined opcodes

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-43
Target: `crypto/func/builtins.cpp`	

Description

The FunC compiler can produce bytecode containing the `NEGPOW2` opcode, which is neither valid Fift nor a documented TVM opcode. For example, the FunC code in Figure 43.1 produces this opcode.

```
1    var foo() {
2       return 1;
3    }
4
5    (int) bar(int b) impure {
6       return b;
7    }
8
9    int main() {
10      _ = bar(-1 << foo());
11      return 0;
12   }
```

Figure 43.1: The left shift of -1 on line 10 generates the NEGPOW2 opcode

Exploit Scenario

A FunC programmer performs a left shift on a constant valued -1, producing the `NEGPOW2` opcode. Fift fails to emit TVM for the resulting code since `NEGPOW2` is not a valid opcode.

Recommendations

Short term, either implement and document the NEGPOW2 opcode, or change FunC's code generation to use a different, valid opcode.

Long term, add differential fuzz testing of FunC to the TON CI pipeline to ensure that no further crashes occur.

44. Invalid syntax can cause the FunC compiler to crash

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-TON-44
Target: `crypto/func/abscode.cpp`	

Description

Consider the (incorrect) FunC code in Figure 44.1. It is a mixture of a variable declaration and function call in one statement. This is not valid FunC syntax, and it should produce a compiler error.

```
 1    #include "../../smartcont/stdlib.fc";
 2    () main() {
 3          var replace0c~udict_replace?(1, 1,
begin_cell().end_cell().begin_parse());
 4    }
```

Figure 44.1: Incorrect FunC code triggering a compiler crash

Instead of producing an error, the func compiler aborts due to an `std::out_of_range` exception not being caught. Line 246 in `Op::split_var_list` (Figure 44.2) is the source of the exception.

```
 243    void Op::split_var_list(std::vector<var_idx_t>& var_list, const
std::vector<TmpVar>& vars) {
 244     int new_size = 0, changes = 0;
 245     for (var_idx_t v : var_list) {
 246       int c = vars.at(v).coord;
 247       if (c < 0) {
 248         ++changes;
 249         new_size += (~c & 0xff);
 250       } else {
```

Figure 44.2: Compiler code throwing the `std::out_of_range`-exception (`crypto/func/abscode.cpp#243–250`)

When invoked, v holds the value -28746, which is not a valid index in the `vars`-vector.

Exploit Scenario

Incorrect FunC code is compiled using func causing a compiler crash.

Recommendations

Short term, locate the root cause of the parse failure and correct it to prevent compiler crashes.

Long term, add automated FunC code generation—both correct and incorrect—to the TON CI pipeline to ensure that no further crashes occur.

---

45. Dictionary lookup can return incorrect results

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TON-45
Target: TVM	

Description

Dictionary queries can return incorrect values, allowing unauthorized actions. Consider the FunC code in Figure 45.1.

```
  1    #include "../../smartcont/stdlib.fc";
  2    int vis(int v) asm "DUMPSTK";
  3    () main() {
  4     var c = new_dict();
  5     c~udict_set(256, 0xb1a5ed00deadbeef, begin_cell().store_uint(1,
1).end_cell().begin_parse());
  6     var (value, success) = c.udict_get?(128, 0);
  7     throw_unless(222, -1 == success);
  8     var uval = vis(value~load_uint(1));
  9     throw_unless(333, 1 == uval);
 10    }
```

Figure 45.1: FunC code illustrating storing a dictionary entry and querying the dictionary for a different key

A 256-bit key is added to the c dictionary on line 5. This is the only key in the dictionary. On the next line, a 128-bit key is queried. This query on line 6 should fail because, regardless of bit lengths, the one and only key stored in the dictionary does not have value zero. Therefore, we would expect line 7 to throw exception 222 because the query should not have resulted in success. However, when the code in Figure 45.1 is compiled and executed, `success` is `true`, indicating that the key exists, and line 7 does not throw an exception. Instead, exception 333 is thrown on line 9, indicating that the stored value is incorrect. At this point, for an `udict_set` using bit length 256, `udict_get?` would report success for key 0 using bit lengths 8, 16, 32, 64, and 128.

The Fift code emitted by the FunC compiler appears to be correct, suggesting that this is an error in the TVM's dictionary implementation.

Exploit Scenario

Consider the FunC code in Figure 45.2. Only certain addresses should be able to invoke `do_owner_action`. However, due to the incorrect bit length used in `udict_get?`, certain

non-present keys would still be considered present. This allows unauthorized invokers to call `do_owner_action`, as shown in Figure 45.2.

```
#include "../../smartcont/stdlib.fc";

() do_owner_action() impure;

() main() {
 var owners = new_dict();
 var owneraddress = 0xb1a5ed00deadbeef;
 owners~udict_set(256, owneraddress, begin_cell().store_uint(1,
1).end_cell().begin_parse());

 ;; ... other code

 var invokeraddress = 0; ;; user controlled
 var (_, ok) = owners.udict_get?(128, invokeraddress);
 throw_unless(333, ok != true);
 do_owner_action();
}
```

Figure 45.2: Vulnerable FunC smart contract code allowing non-intended invokers to call `do_owner_action`

Recommendations

Short term, ensure that only keys actually stored in the dictionary are reported as present.

Long term, implement automated FunC code generation that explores both intended and unintended usage of APIs. If dictionaries are intended to contain only keys of equal bit length, consider setting a dictionary's bit length during construction rather than requiring the bit length on every dictionary operation.

46. Dictionary insertion can inconsistently crash

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-TON-46
Target: Undetermined	

Description

Dictionary insertion is brittle and can inconsistently crash at runtime on valid FunC code. Consider the FunC code in Figure 46.1.

```
1    #include "../../smartcont/stdlib.fc";
2
3    int main() {
4     var val = begin_cell().store_uint(1, 3).end_cell().begin_parse();
5     var key = begin_cell().store_uint(2, 2).end_cell().begin_parse();
6
7     var c = new_dict();
8     c~idict_set(3, 1, val);
9     c~dict_set(2, key, val);
10    return c.dict_empty?();
11   }
```

Figure 46.1: FunC smart contract code setting dictionary key using different key lengths.

When the code is compiled and run, it will throw an `exception code 9: error while parsing a dictionary node label`. However, if an additional line of code is added between lines 8 and 9, such as in Figure 46.2, the program executes without error with `main` returning `false`, indicating that the dictionary is not empty. This type of inconsistent behavior can cause unexpected control flow when executed on-chain.

```
⋮
8     c~idict_set(3, 1, val);
+     c~udict_set(3, 0, val);  ;; <-- this line is inserted
9     c~dict_set(2, key, val);
⋮
```

Figure 46.2: The code from Figure 46.1 with a new line inserted

Exploit Scenario

A TON user deploys a semantically and syntactically correct FunC program similar to Figure 46.1 to mainnet. This contract will unexpectedly crash at runtime.

Recommendations

Short term, document the intended behavior for mixed dictionary operations and ensure the success of the `dict_set?` and related operations is not the result of intermediate calls.

Long term, implement automated FunC code generation that explores both intended and unintended usage of APIs. Consider if the bit length should be a property set during construction of the dictionary, not the individual functions operating on the dictionary.

47. Bitwise negation of false is not always true

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TON-47
Target: `crypto/func/builtins.cpp`	

Description

The ~ (bitwise not) operator does not always result in `true` when applied to `false` variables, which can put funds at risk.

Consider the FunC code in Figure 47.1.

```
1    int func4() {
2           return 12;
3    }
4
5    var main() {
6     var u = (12 == 0);
7     throw_unless(990, u == 0);
8     throw_unless(989, ~ u == -1);
9
10    var z = (func4() == 0);
11    throw_unless(988, z == 0);
12    throw_unless(987, ~ z == -1);
13
14    return 0;
15   }
```

Figure 47.1: FunC code that verifies basic boolean invariants.

It is expected that the u and z variables hold the same value: `false` (equivalent to 0 in FunC). The result of the ~ operator on those variables should equal `true` (equivalent to -1 in FunC). However, when this code is compiled and run, exception 987 is thrown on line 12. This indicates that ~ u is `true`, but ~ z is *not* `true`, despite the fact that they are both equal to zero.

The FunC compiler appears to generate an incorrect, unconditional `THROW 987` when generating code for line 12.

Exploit Scenario

A smart contract deployed on mainnet relies on the bitwise negation of a value. A comparison like that on line 12 of Figure 47.1 returns an incorrect result, affecting control flow in such a way that puts funds at risk.

Recommendations

Short term, ensure that the FunC compiler does not generate incorrect code for the bitwise not operator.

Long term, consider implementing FunC property-based tests that verify that basic properties hold, regardless of whether a value is retrieved as the result of a more complex expression such as a function call or a simple expression.

48. Setting the random number seed from the FunC standard library causes a stack misalignment

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TON-48
Target: `crypto/smartcont/stdlib.fc`	

Description

The definition for the FunC standard library function `set_seed` is shown in Figure 48.1.

```
206    int set_seed() impure asm "SETRAND";
```

Figure 48.1: Definition of the `set_seed` FunC-function (`crypto/smartcont/stdlib.fc#206`)

The function translates directly to the fift SETRAND instruction. According to the TVM-documentation, SETRAND pops the top of the stack and returns nothing. This is inconsistent with the function signature in Figure 48.1.

Exploit Scenario

Code similar to the example in Figure 48.2 is deployed on mainnet as a part of a more complex system.

```
1    #include "../../smartcont/stdlib.fc";
2
3    var main() {
4     var x = set_seed();
5     return x;
6    }
```

Figure 48.2: Simplified deployment scenario invoking `set_seed`

When compiled and executed, this code leaves the stack unaligned with respect to the function declaration. Figure 48.3 shows the trace resulting from running this code.

```
1    implicit PUSH 0 at start
2    execute SETCP 0
3    execute DICTPUSHCONST 19 (xC_,1)
4    execute DICTIGETJMPZ
5    execute SETRAND
6    handling exception code 2: stack underflow
7    default exception handler, terminating vm with exit code 2
8    [ 3][t 0][2022-10-11 07:42:47.764668488][vm.cpp:558]    steps: 5 gas:
```

```
used=362, max=9223372036854775807, limit=9223372036854775807, credit=0
  9    0 2
```

Figure 48.3: Trace from running code in 48.2, showing stack underflow

In a more complex setting, this stack misalignment could potentially consume the incorrect stack value and cause unexpected control flow.

Recommendations

Short term, change the definition of `set_seed` to reflect the actual mechanics of the SETRAND instruction.

Long term, ensure all API functions are properly tested. Consider implementing FunC- and Fift-based tests that verify that changes in stack depth are consistent with respect to the API being invoked.

49. Querying a dictionary throws exception

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TON-49
Target: FunC	

Description
Queries for keys that do not exist in the dictionary throw an exception instead of returning `false`.

Consider the FunC code in Figure 49.1:

```
  1    #include "../../smartcont/stdlib.fc";
  2    () main() {
  3            var c = new_dict();
  4            c~udict_set(16, 1, begin_cell().store_uint(1,
1).end_cell().begin_parse());
  5            var (value, success) = c.udict_get?(9, 1);
  6            throw_unless(222, -1 == success);
  7
  8            var uval = value~load_uint(1);
  9            throw_unless(333, 1 == uval);
 10    }
```

Figure 49.1: FunC code interaction with a dictionary

When compiled and run, this code throws `exception code 10: invalid dictionary fork node`. As there is no key with value 1 of bit length 9 in the dictionary, the `success` should become `false`; but instead the call to `udict_get?` throws.

Exploit Scenario
A token contract is deployed in which the dictionary access during token withdrawal is susceptible to the same error depicted in Figure 49.1. Any tokens transferred to the contract will be permanently stuck, never able to be withdrawn.

Recommendations
Short term, have the dictionary get-methods return `false` when a key is not found instead of throwing an exception. Consider extending the documentation to describe potential erroneous uses of dictionaries.

Long term, implement automated FunC code generation that explores both intended and unintended usage of APIs. Consider if the bit length should be a property set during construction of the dictionary, not the individual functions operating on the dictionary.

50. Compile time integer literal operations can result in unexpected control flow

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-TON-50
Target: FunC	

Description

Consider the FunC code in Figure 50.1. The call to the `visualize` function forces the compiler to emit the value it has computed into Fift code. The corresponding Fift can be found in Figure 50.2.

```
() visualize(int val) impure {
}

() main() {
        var var1 = 13;
        var var2 =
-557313138500893964536174156540965495328613449131902578762062303538626974042 70;
        var v1 = var2 -
996641876494759574118437359129045809906993707312124817323484053310542686648 57;

        visualize(v1);

        var n = v1 | (-7 / var1);
        visualize(n);
}
```

Figure 50.1: FunC code with two large integer literals

```
"Asm.fif" include
// automatically generated from `../crypto/func/opttest/intlit.fc`
PROGRAM{
 DECLPROC visualize
 DECLPROC main
 visualize PROC:<{
   DROP
 }>
 main PROC:<{
   PUSHNAN
   DUP
   visualize CALLDICT
   DROP
   -1 PUSHINT
```

```
15      visualize CALLDICT
16     }>
17    }END>c
```

Figure 50.2: Fift program corresponding to FunC code in Figure 50.1

First, on line 10 of Figure 50.2, it is evident that v1 is out of range, as it is a PUSHNAN instruction. However, on line 14 of Figure 50.2, an expression resulting from bitwise or with v1 and another expression results in an actual number, -1. It is unexpected that an operation with NaN does not result in another NaN.

Further, consider the FunC code in Figure 50.3 and its compiled Fift code in Figure 50.4.

```
1    () visualize(var val) impure {
2    }
3
4    () main() {
5     visualize((-1) >> 0);
6     visualize((-1) >> 255);
7     visualize((-1) >> 256);
8     visualize((-1) >> 257);
9    }
```

Figure 50.3: FunC code exercising a right shift operation

```
1    "Asm.fif" include
2    // automatically generated from `../crypto/func/opttest/shiftr.fc`
3    PROGRAM{
4     DECLPROC visualize
5     DECLPROC main
6     visualize PROC:<{
7       DROP
8     }>
9     main PROC:<{
10      -1 PUSHINT
11      visualize CALLDICT
12      -1 PUSHINT
13      visualize CALLDICT
14      -1 PUSHINT
15      visualize CALLDICT
16      PUSHNAN
17      visualize CALLDICT
18    }>
19   }END>c
```

Figure 50.4: Fift code resulting from the FunC code in Figure 50.3.

On line 10 in Figure 50.4, an expected -1 is the result of (-1) >> 0. On lines 12 and 14, the results of larger integer shifts are visible. From Figure 50.4, it is evident that the full 257 bits can be shifted right, beyond which the result becomes NaN.

Two right-shift operators are defined in section 5.3 of the documentation for TVM instructions. One of them is defined as a maximum shift of 256 bit positions, and the other as the maximum shift for 1024 bit positions.

In this case, it is not clear which operation is actually implemented in the FunC compiler unless the resulting Fift code is manually inspected.

Exploit Scenario

An unsuspecting developer writes FunC code that during compile time is translated into values the developer did not anticipate, causing unexpected control flow at runtime.

Recommendations

Short term, ensure that code in Figure 50.2 and Figure 50.3 is translated as expected or correct it.

Long term, document how the FunC compiler handles integer literal computations at compile time. For reference, the Solidity documentation on literal integer handling.

52. Ethereum bridge signature verification will always pass for address zero

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-TON-52
Target: `bridge-solidity/contracts/SignatureChecker.sol`	

Description

The TON Ethereum bridge validates signatures using the `ecrecover` function (see Figure 52.1).

```
41    require(ecrecover(prefixedHash, v, r, s) == sig.signer, "Wrong signature");
```

Figure 52.1: The Solidity `ecrecover` function is used to validate signatures. (contracts/SignatureChecker.sol#41)

The `ecrecover` function returns zero on failure, so if `sig.signer == 0`, then any signature will be accepted regardless of whether or not it is cryptographically valid.

This finding is informational because this function is currently called only when using `msg.sender`, which is validated to have been an oracle before the code in Figure 52.1 is executed (see Figure 52.2).

```
25    require(isOracle[signer], "Unauthorized signer");
```

Figure 52.2: `sig.signer` is implicitly verified to be an oracle. (contracts/Bridge.sol#25)

Therefore, this will currently be an issue only if the zero address is ever added as an oracle. However, if the signature validation function is ever used from a different code path, or if an attacker discovers how to set the zero address as an oracle (or somehow bypass the check in Figure 52.2), this would be a critical finding.

Exploit Scenario

An attacker discovers a way to bypass the check in Figure 52.2, allowing them to execute fraudulent bridge transfers with invalid signatures.

Recommendations

Short term, add documentation to the Signature Checker contract to warn future developers that all signatures from the zero address will be accepted, regardless of whether or not they are valid.

Long term, consider adding an additional check during signature validation to ensure that the signer is not the zero address.

53. Context sensitivity of the ; token can lead to confusion and bugs

Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-TON-53
Target: FunC	

Description

The “;” token is used both as an end-of-line delimiter (similar to the majority of programming languages) and to denote the beginning of a single-line comment (similar to Lisp). Since inter-token whitespace is significant in FunC, an errant space between two “;” tokens (see line 3 of Figure 53.1 for an example) can silently turn what was intended to be a comment into an actual statement.

```
1    int main() {
2     var v1 = 12;
3     ; ; v1 = 0;
4     return v1 == 0;
5    }
```

Figure 53.1: Line 3 will be treated as a statement rather than a comment.

Exploit Scenario

A FunC programmer intends to comment out an old statement, as in line 2 of Figure 53.2.

```
1    int main() {
2     var v1 = 12;  ; ; v1 = 0;
3     return v1 == 0;
4    }
```

Figure 53.2: v1 will be reassigned value zero since there is whitespace between the last two semicolons on line 2.

Since there is whitespace between the last two semicolons on line 2, v1 will be reassigned value zero, which is not what the programmer intended.

Recommendations

Short term, consider adding a compiler warning to FunC when there is an empty statement (i.e., an unnecessary end-of-statement “;” delimiter). The last two semicolons on line 2 of Figure 53.2 would ideally each produce such a warning.

Long term, consider deprecating the use of semicolons as single-line comments and switching to a less ambiguous token, like "`//`". Also consider defining a formal grammar for FunC so (a) the parser can be automatically generated from the grammar, and (b) other tools can be developed to correctly parse FunC (e.g., linters and static analyzers that can detect bugs like this).

Summary of Recommendations

The TON TVM and Fift scripting language are in active development. Trail of Bits recommends that TON address the findings detailed in this report and take the following additional steps:

- Integrate automated linting using tools like Cppcheck (see Appendix E) into the TON continuous integration pipeline.
- Regularly fuzz test the codebase, particularly all entry points that accept untrusted user input.
- Improve unit tests to cover all TVM opcode families.
- Regularly run all unit and fuzz tests with LLVM sanitizers enabled (see Appendices D and G).
- Improve inline comments in the codebase.
- Implement integration tests that do not depend on third-party software (i.e., MyLocalTon).

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
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
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. Code Quality Recommendations

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

General recommendations

- **Ensure that all classes obey the "rule of five."** Every C++ class that implements a custom destructor, copy-constructor, copy-assignment operator, move constructor, or move assignment operator should implement all five. For example, CellBuilder implements only three of the five:
-

```
32    class CellBuilder : public td::CntObject {
 ⋮
48      CellBuilder();
49      virtual ~CellBuilder() override;
 ⋮
93      CellBuilder& operator=(const CellBuilder&);
94      CellBuilder& operator=(CellBuilder&&);
```

Figure C.1: CellBuilder does not obey the "rule of five"

For more information, see:

- https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rc-five
- https://en.cppreference.com/w/cpp/language/rule_of_three

- **Use `std::move` only when absolutely necessary.** The TON codebase includes many uses of `std::move` that are at best redundant, can sometimes prevent compiler optimizations, and at worst can lead to security findings like TOB-TON-1. The codebase has 186 usages of `std::move` to return a value from a function. These are all unnecessary, and will in fact prevent the compiler from performing Named Return Value Optimization (NRVO), which would produce even more performant code than the `std::move`. You can detect such unnecessary moves by adding the `-Wpessimizing-move` and `-Wredundant-move` compiler options.

- **Ensure the system can be built with Address Sanitizer enabled.** The `CMakeLists.txt` file includes an option to enable Address Sanitizer. The system cannot be built when Address Sanitizer is enabled because it detects leaks that interfere with the build process. This applies to, for example, `test-vm`, `test-smartcont`, and `test-fift`. Use of Address Sanitizer is highly recommended, but given the current state of the build, it is hard to identify actual

leaks or memory corruptions. Our primary recommendation is to fix memory leaks or to suppress them using ASAN suppressions.

- **The func- and fift-programs leak memory.**
  When running `func` and `fift` with Address Sanitizer enabled, memory leaks are reported, as shown in the example in Figure C.2. If that code is executed using `fift`, Address Sanitizer reports leaks.

```
recursive append-long-bytes {
  over Blen over <= { drop B, } {
    B| <b swap 127 append-long-bytes b> -rot B, swap ref,
  }  cond
} swap !
```

Figure C.2: Fift code that triggers leak detection

  To help narrow down the source of leaks, the `lsan_do_recoverable_leak_check()`-function can be invoked at different points of a program; memory leaks are detected upon destruction. We used this approach to detect that the class `fift::Fift` is responsible for memory leaks.

  For the func binary, memory leaks are detected related to the `Expr` class. The source code suggests these leaks are intentional. We recommend fixing the memory leaks using known patterns, such as `std::shared_ptr` or the `Ref`- and `Cnt`-templates used elsewhere in the TON-blockchain. Alternatively, we recommend suppressing known leaks using ASAN suppressions to avoid missing any unintended leaks.

- **Use of deprecated openssl-functions.**
  When compiling the `tdutils` library, compiler warnings about deprecated functions are emitted. Among them are the functions `MD5`, `AES_cbc_encrypt`, and `AES_set_decrypt_key`. Although we were not able to attribute any security issue to the use of deprecated functions, we recommend replacing them with the recommended, non-deprecated versions to prevent any future issues.

- **FunC double negation expressions fail to compile.**
  Consider the code in Figure C.3.

```
int main() {
 return ~ - 1;
}
```

Figure C.3: FunC code with double negation

  When compiling using func, the code fails with the message `error: identifier expected instead of `-``. However, if the expression is rewritten as `~ (- 1);`,

it compiles. This indicates a potential expression parsing issue with repeated unary expressions. We recommend fixing this and adding additional test cases to ensure that similar uncommon patterns can also be compiled. Failure to compile the code in Figure C.3 could indicate a more serious underlying problem.

- **Prefer range-based for loops and STL algorithms over explicit indexing.** Consider the code in Figure C.4.

```
114    for (size_t idx = 0; idx < in_desc_.size(); idx++) {
115     if (in_desc_[idx] == desc) {
116       in_desc_[idx].cat_mask |= desc.cat_mask;
117       return;
118     }
119    }
```

Figure C.4: Loop with explicit index (adnl/adnl-network-manager.hpp#114–119)

We recommend using range-based `for` instead of index-based loops to clarify intent. For the particular case described here, using `std::find` would make the purpose clear. Our general recommendation is to prefer range-based `for` over explicit indexing and named algorithms over range-based `for`, as applicable.

- **Throw by value, catch by (const) reference.**
When using C++ exceptions, we recommend throwing by value and catching by reference to const. This prevents issues with object slicing in inheritance hierarchies. There are a number of locations in the code base where exceptions are caught by value, such as block.cpp, mc-config.cpp, and check-proof.cpp. Catching by value could also be an issue when the copy constructor of the exception class throws, as that will invoke `std::terminate()`.

- **Ensure that all test cases can be successfully run in CI.**
When running the test-smartcont test binary, it will exit due to `[0][t 0][2022-07-15 07:55:14.058572432][test-smartcont.cpp:1197] Check ``manual.write().send_external_message(set_query).code == 0`` failed`.
Additionally, several of the test case files in the `crypto/func/test` directory fail to compile, such as the `a6_2.fc` test file.
We recommend enforcing successful execution of test cases as part of CI to prevent code quality and test coverage degradation over time.

`crypto/vm/dict.cpp`:

- **Do not call virtual methods during object construction.** If the `validate` argument to any of the constructors of `DictionaryBase` is set to `true`, the

`force_validate()` function will be invoked. This will in turn invoke the virtual method `validate()`.

Invoking the base-class version of a virtual method during object construction might not work as expected; see https://isocpp.org/wiki/faq/strange-inheritance#calling-virtuals-from-ctors.

`AugmentedDictionary` appears to override the `validate` method, but it handles the situation by invoking `force_validate()` in each of its own constructors.

`crypto/vm/cells/CellUsageTree.cpp`:

- **Consider passing arguments by reference if they can never be `nullptr`.** In the `mark_path` function of the cell usage tree node, if the `master_tree` argument is ever `nullptr` and the build has `NDEBUG` defined, then there will be a null pointer dereference on line 57. Passing `master_tree` as `CellUsageTree&` would prevent this at compile time.

```
 51    bool CellUsageTree::NodePtr::mark_path(CellUsageTree* master_tree)
const {
 52      DCHECK(master_tree);
 53      auto tree = tree_weak_.lock();
 54      if (tree.get() != master_tree) {
 55        return false;
 56      }
 57      master_tree->mark_path(node_id_);
 58      return true;
 59    }
```

Figure C.5: The `master_tree` pointer can be changed to a reference

This function does not appear to ever be called in a context where `master_tree` could be `nullptr`, but it could be added in the future.

`validator-session/validator-session-state.cpp`:

- **Dereferencing a null pointer is undefined behavior.** Several log messages (e.g., line 1427) either explicitly or implicitly (via the << operator override on lines 60–62) dereference the `action` pointer, which can be null. Explicitly check whether `action` is `nullptr` before dereferencing it.

`crypto/test/modbigint.cpp`:

- **Ensure the required C++ version is aligned with the features used.** In `CMakeLists.txt`, the required C++ version is set to 14.

```
82    set(CMAKE_CXX_STANDARD 14)
83    set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
84    set(CMAKE_CXX_EXTENSIONS FALSE)
```

Figure C.6: C++ version configuration (CMakeLists.txt#82–84)

However, in several locations of `modbigint.cpp`, `static_assert` is used without a message. Use of `static_assert` without a message is a C++17 feature.

```
78    static_assert(M >= N);
```

Figure C.7: Use of C++ 17 feature (crypto/test/modbigint.cpp#78)

`crypto/smartcont/stdlib.fc`:

- **FunC standard library calls that implicitly end cells are unintuitive.** For example, the FunC standard library function `~udict_set_builder` resolves to the Fift opcode `DICTUSETB`, which implicitly adds an `ENDC` operation. In usages of standard library functions like `~udict_set_builder` (e.g., here), the presence of `begin_cell()` but lack of an associated `end_cell()` will look like a bug to a FunC programmer who is not also familiar with the entire TVM instruction set. This could be made more clear by renaming `begin_cell()` to `create_cell_builder()`, because that is closer to the semantics of what the command is actually doing.

`crypto/vm/box.hpp`:

- **The Box class has a single mutable member but all methods are const.** It is not clear why this is the case. If there is a compelling reason for this pattern, document it in the code. If not, consider making the `data_` variable a regular member and remove the `const` qualifier from all methods that mutate it.

`crypto/func/analyzer.cpp`:

- **Intentional use of bit-wise OR to avoid short-circuiting should be documented.** Several usages of the bit-wise OR operator—particularly in this file, but also throughout the codebase—eschew the logical `OR` operator, presumably to require the side effects of the right-hand argument that would otherwise be short-circuited if the left-hand argument were false. This behavior should be documented to prevent future developers from converting the bitwise operators to logical operators.

D. Risks of Undefined Behavior in C++

The C++ standard imposes no restrictions on the observable operation of a program that executes undefined behavior, such as accessing memory outside of array bounds, null pointer dereferencing, signed integer overflow, and bit-shifting by negative values. Although a program is capable of operating normally even if it executes undefined behavior, there is no guarantee of this. In fact, most compilers can and will silently break programs containing undefined behavior in subtle, hard-to-catch ways, particularly when applying optimizations.

Examples of Undefined Behavior

For example, consider the following program that has a negative bit-shift on line 3:

```
1    int main(int argc, char** argv) {
2        if(argc > 1) {
3            return 1234 << -2;
4        } else {
5            return 0;
6        }
7    }
```

Figure B.1: A simple program that exhibits undefined behavior on line 3

With optimizations enabled, the latest version of the `clang` compiler will correctly identify the undefined behavior on line 3 and completely optimize out the entire first half of the branch. The resulting assembly for the compiled program—that always returns zero regardless of the inputs—is given in Figure B.2.

```
1    main:                                   # @main
2            xorl    %eax, %eax
3            retq
```

Figure B.2: The assembly listing for the program in Figure B.1 compiled with optimizations.

A more insidious example of the dangers of undefined behavior is given in Figure B.3, below:

```
1   #include <limits>
2   #include <cstdint>
3   #include <iostream>
4   int main(int argc, char *argv[]) {
5     uint32_t u0 = std::numeric_limits<uint32_t>::max();
6     uint32_t u1 = u0 + 1;
7
8     if (u1 < u0) {
9       std::cout << "Unsigned wrap!" << std::endl;
10    }
11    std::cout << "u0: " << u0 << " u1: " << u1 << std::endl;
12
13    int32_t i0 = std::numeric_limits<int32_t>::max();
14    int32_t i1 = i0+1;
15
16    if (i1 < i0) {
17      std::cout << "Signed wrap!" << std::endl;
18    }
19    std::cout << "i0: " << i0 << " i1: " << i1 << std::endl;
20  }
```

Figure B.3: A real-world example of the dangers of undefined behavior

When compiled without optimizations enabled, the code will print

```
Unsigned wrap!
u0: 4294967295 u1: 0
Signed wrap!
i0: 2147483647 i1: -2147483648
```

as would be expected.

However, line 14 contains a signed integer overflow, which is undefined behavior. With optimizations enabled, `clang` will optimize away the entire if statement on lines 16 through 18 and instead print

```
Unsigned wrap!
u0: 4294967295 u1: 0
i0: 2147483647 i1: -2147483648
```

How to Detect Undefined Behavior

Although some types of undefined behavior can be caught at compile time by static analyzers like cppcheck and clang-tidy, most undefined classes of behavior are highly dependent on runtime context. Clang and gcc both have undefined behavior sanitizers (ubsan) that can instrument the code to report when the program encounters

undefined behavior during execution. We recommend running all unit and fuzz tests with ubsan enabled.

E. Automated Static Analysis

This appendix describes the setup of the automated analysis tools used in this audit.

Though static analysis tools frequently report false positives, they detect certain categories of issues with essentially perfect precision, such as memory leaks, misspecified format strings, and use of unsafe APIs. We recommend that you periodically run these static tools and review their findings.

Cppcheck

To install Cppcheck, we followed the instructions on the official website. We ran the tool with all analyses enabled:

```
cppcheck --enable=all --inconclusive . 2> cppcheck.txt
```

The tool helped us to find the issue described in TOB-TON-4 as well as some of the issues described in the code quality appendix.

F. Automated Dynamic Analysis

This appendix describes the setup of the automated dynamic analysis tools and test harnesses used during this audit.

In most software, testing typically includes only unit and integration tests. These types of tests detect the presence of functionality that adheres to the expected specification. However, they do not account for other potential behaviors that an implementation may have.

Fuzzing and property testing complement both unit and integration testing through the identification of extra behavior in a component of a system. Test cases are generated and subsequently provided to a component of the system as input. Upon execution, properties of the component are observed for deviations from expected behaviors.

The primary difference between fuzzing and property testing is the method of generating inputs and observing behavior. Fuzzing typically attempts to provide random or randomly mutated inputs in an attempt to identify edge cases in entire components. Property testing typically provides inputs sequentially or randomly within a given format, checking to ensure a specific property of the system holds upon each execution.

By developing fuzzing and property testing alongside the traditional set of unit and integration tests, the overall security posture and stability of a system is likely to improve since edge cases and unintended behaviors can be pruned during the development process.

libFuzzer-Based Test Cases for TON

We have included a collection of fuzz tests that uses libFuzzer, an in-process, coverage-guided, evolutionary fuzzing engine integrated into Clang. These tests cover a variety of deserialization and processing functions, as well as functions that handle untrusted inputs. We integrated them into the build process to improve the coverage of the TVM and Fift code. For instance, figures F.1 and F.2 show the libFuzzer tests that we created to automatically generate both valid and invalid TVM opcode sequences.

```
#include <algorithm>

#include "vm/vm.h"
#include "vm/cp0.h"
#include "vm/dict.h"
#include "td/utils/tests.h"

std::string run_vm(td::Ref<vm::Cell> cell) {
       vm::init_op_cp0();
       vm::DictionaryBase::get_empty_dictionary();
```

```
	class Logger: public td::LogInterface {
	public:
		void append(td::CSlice slice) override {
			res.append(slice.data(), slice.size());
		}
		std::string res;
	};
	static Logger logger;
	logger.res = "";
	td::set_log_fatal_error_callback([](td::CSlice message) {
		td::default_log_interface->append(logger.res);
	});
	vm::VmLog log { &logger, td::LogOptions::plain() };
	log.log_options.level = verbosity_FATAL;
	log.log_options.fix_newlines = true;
	td::set_verbosity_level(verbosity_PLAIN);
	auto total_data_cells_before = vm::DataCell::get_total_data_cells();
	SCOPE_EXIT {
		auto total_data_cells_after = vm::DataCell::get_total_data_cells();
		ASSERT_EQ(total_data_cells_before, total_data_cells_after);
	};

	vm::Stack stack;
	vm::GasLimits gas_limit(1000, 1000);

	vm::run_vm_code(vm::load_cell_slice_ref(cell), stack, 0 /*flags*/,
			nullptr /*data*/, std::move(log) /*VmLog*/, nullptr,
&gas_limit);
	return logger.res;  // must be a copy
}

td::Ref<vm::Cell> to_cell(const unsigned char *buff, int bits) {
	return vm::CellBuilder().store_bits(buff, bits, 0).finalize();
}

/* run_vm_code */
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
	run_vm(to_cell(Data, std::min(Size*8, static_cast<size_t>(1023))));
	return 0;
}
```

Figure F.1: A libFuzzer test for running automatically generating possibly invalid TVM opcode sequences.

```
/*
 * vm_instr_fuzz.cpp
 *
 *  Created on: 14 Jul 2022
 *      Author: hbrodin
 */
```

```
#include <algorithm>

#include "vm/vm.h"
#include "vm/cp0.h"
#include "vm/dict.h"
#include "td/utils/tests.h"

std::string run_vm(td::Ref<vm::Cell> cell) {
	vm::init_op_cp0();
	vm::DictionaryBase::get_empty_dictionary();

	class Logger: public td::LogInterface {
	public:
		void append(td::CSlice slice) override {
			res.append(slice.data(), slice.size());
		}
		std::string res;
	};
	static Logger logger;
	logger.res = "";
	td::set_log_fatal_error_callback([](td::CSlice message) {
		td::default_log_interface->append(logger.res);
	});
	vm::VmLog log { &logger, td::LogOptions::plain() };
	log.log_options.level = verbosity_FATAL;
	log.log_options.fix_newlines = true;
	td::set_verbosity_level(verbosity_PLAIN);
	auto total_data_cells_before = vm::DataCell::get_total_data_cells();
	SCOPE_EXIT {
		auto total_data_cells_after = vm::DataCell::get_total_data_cells();
		ASSERT_EQ(total_data_cells_before, total_data_cells_after);
	};

	vm::Stack stack;
	vm::GasLimits gas_limit(1000, 1000);

	vm::run_vm_code(vm::load_cell_slice_ref(cell), stack, 0 /*flags*/,
			nullptr /*data*/, std::move(log) /*VmLog*/, nullptr,
&gas_limit);
	return logger.res;  // must be a copy
}

td::Ref<vm::Cell> to_cell(const unsigned char *buff, int bits) {
	return vm::CellBuilder().store_bits(buff, bits, 0).finalize();
}


void serialize(const uint8_t *data, size_t size) {

	size_t consumed = 0;
	size_t nfinalized = 0;
	std::vector<td::Ref<vm::Cell>> cells;
```

```
	while (consumed < size) {
		auto avail = size-consumed;
		auto avail_bits = avail*8;
		//auto consume_bits = std::min(avail_bits, 1023ul);
		auto consume_bits = std::min(avail_bits, 257ul);
		auto consume_bytes = consume_bits/8+1; // roughly...

		vm::CellBuilder cb;
		cb.store_bits(data + consumed, consume_bits, 0);

		bool stop = false;
		if (nfinalized >= vm::Cell::max_refs) {
			for (size_t ci =
nfinalized-vm::Cell::max_refs;ci<nfinalized;ci++) {
				if (!cb.store_ref_bool(cells[ci])) {
					stop = true;
					break;
				}
			}
		}
		if (stop)
			break;
		if (cb.get_depth() > vm::Cell::max_depth)
			break;
		cells.push_back(cb.finalize());
		nfinalized++;
		consumed += consume_bytes;
	}
	if (cells.empty())
		return {};
	return cells.back();
}

/* run_vm_code_specific */
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
	auto cells = to_cells(Data, Size);
	if (!cells)
		return -1;
	run_vm(*cells);
	return 0;
}
```

Figure F.2: A libFuzzer test for running automatically generating valid TVM opcode sequences.

These tests cover the following functionality:

- Feeds randomly generated cells to `Vm::run_vm_code` to uncover memory safety, undefined behavior, and abrupt termination errors.

- Feeds randomly generated cells containing valid instructions to `Vm::run_vm_code` to uncover memory safety, undefined behavior, and abrupt termination errors.

Setting Up the Tests

To build the libFuzzer tests, we recommend using Clang++ version 10.0 or newer. The `CXXFLAGS` variable will need to be modified in the makefile to include the `-fsanitize=fuzzer,address,undefined` flag. This flag will enable the fuzzer as well as the AddressSanitizer and UndefinedBehaviorSanitizer detectors to catch subtle issues that may not cause the program crash.

Measuring Coverage

Regardless of how inputs are generated, an important task after running a fuzzing campaign is to measure its coverage. To do so, we used Clang's source-based code coverage feature. This feature can be enabled by adding the `--enable-cov` flag to the `CXXFLAGS` variable. We recommend keeping a separate build to measure coverage because this flag could be incompatible with the libFuzzer instrumentation.

Integrating Fuzzing and Coverage Measurement into the Development Cycle

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

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

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

Designing Testable Systems

Modern software development best practices typically lead to easier implementation of fuzzing and property testing. System modularity, use of reusable libraries, a centralized configuration system, and isolated execution all help ease the development of testing harnesses.

By forming a system of modular components, each component can be tested independently. This typically reduces the complexity of each component's test harness, as well as helps improve the overall efficiency of testing since test coverage can usually be more easily achieved through independent configuration, and expensive-to-test components do not affect the testing of other components.

Compounding the use of modular components, reusing libraries helps improve test coverage, since the libraries themselves can be tested directly. For example, if an application uses a function defined in such a library, but the path required to gain coverage of the function is difficult for the test harness to reach, this is not as much of a concern since the function is independently testable. This applies to all components that re-use these libraries.

Identifying Properties and Choosing Their Test Methods

To make fuzzing and property testing effective, it's important to choose the appropriate testing method and baseline properties for expected behaviors. This process varies depending on the target, but the same general approach applies.

Evaluating the expected behaviors of a system is often an easy way to identify properties to test. For example, consider a marketplace application that allows users to purchase listed items in bulk through a JSON API. Properties to test might include:

- Users should only be able to submit orders in valid JSON to the API.
- Users should not be able to view a listing if the supply is 0.
- Users should not be able to purchase more than the available supply.

Given these properties of the system, we can evaluate which properties would be most suitable for fuzzing. For the first property, we are evaluating the correctness of the API's JSON parser for potential flaws that could lead to the malicious parsing of invalid JSON. A fuzzer is likely the best approach for this property since it is targeting parser logic, which typically involves mutating inputs over time either randomly or sequentially to gain path coverage.

The remaining properties extend deeper into the system, beyond the parsing of the JSON. In this case, we know the format of the order JSON, and want to test how the parameters of an order affect our properties. Therefore, property testing is likely the best approach. We

can build property tests to ensure these properties hold before, during, and after all interactions with the API. Conditions for these might be as follows:

- Users should not be able to view a listing if the supply is 0.
  - If `listing.visible == true` *and* `listing.supply > 0`
    - The listing is visible with available supply.
  - If `listing.visible == false` *and* `listing.supply == 0`
    - The listing is not visible and has no available supply.
- Users should not be able to purchase more than the available supply.
  - If `listing.supply <= listing.initial_supply`
    - The listing supply has not exceeded the initial supply.

Given property tests for these conditions, potential issues—such as if `listing.supply` is defined as a `uint`, with facile order validations such as `(listing.supply - order.amount) > 0 ? listing.fulfill(order) : listing.deny(order)`—could result in a situation such as `(10 - 11) > 0` evaluating to `true` due to unsigned integer underflow. This could lead to subsequent validations failing to apply, influencing `listing.visible` and `listing.supply` and resulting in undefined behavior.

Automated FunC Test Case Generation

We used two similar but slightly different automated techniques to detect issues in FunC compilation. Both methods are based on automatically generated FunC source code, which is then compiled and run. The first method we employed aimed at finding optimization differentials, in which the generated code differs depending on the selected optimization level in the FunC compiler. This has previously been a problem for other blockchains (1, 2, 3). The second method attempts to validate the generated code according to a model. Both approaches are described in the following subsections.

The benefit to using these techniques is that combinations of code patterns, both sensical and nonsensical, are rapidly tested. As demonstrated in Figure F.3. and Figure F.4, this is evidently a start. Additionally, both techniques can easily be extended (and combined into one) to cover more of the FunC code generation.

Differential testing by optimization level

The Python code in Figure F.3 illustrates the process we used to detect optimization differentials. The overall concept is to construct an expression (in this case, very basic expressions), then compile it using different optimization levels. Finally, in order to confirm that the results are equal, the compiled target functions are glued together using Fift code that ensures the two different functions evaluate to the same value.

During evaluation, additional data (e.g., log files from Undefined Behavior Sanitizer and Address Sanitizer) is collected. This allows the automatic identification of additional issues.

The code in Figure F.3 was used to detect TOB-TON-30.

```
  1  import glob
  2  import os
  3  import random
  4  import subprocess
  5
  6  from itertools import combinations
  7  from pathlib import Path
  8
  9  func = "./crypto/func"
 10  fift = "./crypto/fift"
 11  fift_inc = "../crypto/fift/lib/"
 12  ineqality_error = 999
 13
 14  ubsan_base = "ubsan.log*"
 15  eval_func_name = "eval"
 16
 17
 18  def gen_expr(vars, ops, nops):
 19    ints = [str(random.randint(-2, 10)) for x in vars]
 20    operands = vars + ints
 21    selected_vars = random.choices(operands, k=nops)
 22    selected_ops = random.choices(ops, k=nops)
 23    return " ".join([x for t in zip(selected_vars, selected_ops) for x in
t][:-1])
 24
 25
 26  def gen_res(i, vars, unops, ops, assign_ops, nops):
 27    varname = f"result{i}"
 28    expr = gen_expr(vars, ops, nops)
 29    pre = random.choice(unops + [""])
 30    expr = f"{pre} {expr}"
 31
 32    if random.randint(0, 5) == 1:
 33      # Choose any of the existing variables and reassign it
 34      v = random.choices(vars, k=1)[0]
 35      aop = random.choices(assign_ops, k=1)[0]
 36      decl = f"{v} {aop} {expr};"
 37      return (None, decl)
 38    else:
 39      # Create a new variable and assign to it
 40      decl = f"var {varname} = {expr};"
 41      return (varname, decl)
 42
 43
 44  def gen_eval_func(nlines, nops):
 45    v = ["l", "r"]
```

```
  46      binops = ["*", "/", "~/", "^/", "%", "&", "+", "-", "|", "^", "<<", ">>",
"~>>", "^>>"] #, "==", "!=", "<", "<=", ">", ">=", "<=>"]
  47      unops = ["~", "-"]
  48      assign_ops = ["+=", "-=", "*=", "/=", "~/=", "^/=", "%=", "~%=", "^%=",
"<<=", ">>=", "~>>=", "^>>=", "&=", "|=", "^="]
  49      ret = f"int {eval_func_name}(int l, int r) {{\n"
  50      for i in range(0,nlines):
  51        (n, p) = gen_res(i, v, unops, binops, assign_ops, nops)
  52        if n:
  53          v.append(n)
  54        ret += f"  {p}\n"
  55      return ret + f"  return {v[-1]};\n}}\n"
  56
  57
  58    def gen_eval_fc(working_dir, nlines, nops):
  59      evfc = working_dir / "eval.fc"
  60      with open(evfc, "w") as f:
  61        f.write(gen_eval_func(nlines, nops))
  62      return evfc
  63
  64
  65    def gen_eval_fif(eval_fc, working_dir, optlevel):
  66      tmpfif = working_dir / "tmp.fif"
  67      ofif = working_dir / f"eval0{optlevel}.fif"
  68      try:
  69        subprocess.check_call([func, "-o", tmpfif, f"-O{optlevel}", eval_fc])
  70        with open(tmpfif, "r") as f:
  71          s = f.read()
  72          with open(ofif, "w") as fw:
  73            fw.write(s.replace(eval_func_name, f"{eval_func_name}0{optlevel}"))
  74          return ofif
  75      except:
  76        return None
  77
  78
  79    def archive_file(i, src, target_dir):
  80      if src.exists():
  81        dst_name = target_dir / f"{src.name}.{i}"
  82        os.rename(src, dst_name)
  83
  84
  85    def locate_ubsan_file(working_dir, base=ubsan_base):
  86      l = glob.glob(base, root_dir=working_dir)
  87      if l:
  88        return working_dir / l[0]
  89      return None
```

```
90
91
92    def drop_ubsan_files(working_dir, base=ubsan_base):
93      for f in glob.glob(base, root_dir=working_dir):
94        os.unlink(working_dir / f)
95
96
97    def archive_files(i, working_dir: Path, target_dir):
98      print(f"Error detected, archiving to {target_dir} using i: {i}")
99      archive_file(i, working_dir / "eval.fc", target_dir)
100     archive_file(i, working_dir / "comparison1.fif", target_dir)
101     archive_file(i, working_dir / "comparison2.fif", target_dir)
102     archive_file(i, working_dir / "comparison_error.txt", target_dir)
103     ubsan = locate_ubsan_file(working_dir)
104     if ubsan:
105       archive_file(i, ubsan, target_dir)
106
107
108   def gen_comparison_fif(i, working_dir, o0, o2, argl=1, argr=2):
109     with open(o0, "r") as fo0:
110       o0src = fo0.read()
111     with open(o2,"r") as fo2:
112       o2src = fo2.read()
113
114     fif = (f'"Asm.fif" include\n'
115       f'PROGRAM{{\n'
116       f'{o0src}\n'
117       f'{o2src}\n'
118       f'DECLPROC main\n'
119       f'main PROC:<{{\n'
120       f'\t{argl} PUSHINT\n'
121       f'\t{argr} PUSHINT\n'
122       f'\t{eval_func_name}O0 CALLDICT\n'
123       f'\t{argl} PUSHINT\n'
124       f'\t{argr} PUSHINT\n'
125       f'\t{eval_func_name}O2 CALLDICT\n'
126       f'\tEQUAL\n'
127       f'\t{ineqality_error} THROWIFNOT\n'
128       f'}}>\n'
129       f'}}END>s\n'
130       f'dup\n'
131       #f'dup csr.\n'
132       f'runvmdict .s\n'
133     )
134
135     dst = working_dir / f"comparison{i}.fif"
```

```
136       with open(dst, "w") as dstf:
137         dstf.write(fif)
138       return dst
139
140
141     def run_comparison(compare_fif):
142       return subprocess.run([fift, "-I", fift_inc, compare_fif],
stdout=subprocess.PIPE, stderr=subprocess.PIPE)
143
144
145     def run_comparison_both_ways(working_dir, o0fif, o2fif, l, r):
146       compfif1 = gen_comparison_fif(1, working_dir, o0fif, o2fif, l, r)
147       compfif2 = gen_comparison_fif(2, working_dir, o2fif, o0fif, l, r)
148
149       ret1 = run_comparison(compfif1)
150       ret2 = run_comparison(compfif2)
151
152       def _dump_ret(r1, r2):
153         dstfile = working_dir / "comparison_error.txt"
154         with open(dstfile, "w") as f:
155           f.write(f"r1.returncode: {r1.returncode}\n")
156           f.write(f"r2.returncode: {r2.returncode}\n")
157
158           f.write(f"r1.stdout: {str(r1.stdout, 'utf-8')}\n")
159           f.write(f"r2.stdout: {str(r2.stdout, 'utf-8')}\n")
160
161           f.write(f"r1.stderr: {str(r1.stderr, 'utf-8')}\n")
162           f.write(f"r2.stderr: {str(r2.stderr, 'utf-8')}\n")
163         return dstfile
164
165       if ret1.returncode != ret2.returncode:
166         return _dump_ret(ret1, ret2)
167       if ret1.stdout != ret2.stdout:
168         return _dump_ret(ret1, ret2)
169       if bytes(f"0 {ineqality_error}", "utf-8") in ret1.stdout:
170         return _dump_ret(ret1, ret2)
171
172       # OK!
173       return None
174
175
176     def main():
177       working_dir = Path(os.getcwd()) / "compile_eval"
178       if not working_dir.exists():
179         os.mkdir(working_dir)
180
```

```
181     func_compile_fails = working_dir / "func_fail"
182     if not func_compile_fails.exists():
183       os.mkdir(func_compile_fails)
184
185     ubsan_dir = working_dir / "ubsan"
186     if not ubsan_dir.exists():
187       os.mkdir(ubsan_dir)
188
189     fift_run_fails = working_dir / "fift_fail"
190     if not fift_run_fails.exists():
191       os.mkdir(fift_run_fails)
192
193     print(f"Working dir is {working_dir}")
194
195     i=0
196
197     inputs = combinations([-1, -2, -100000, 0, 1, 2, 100000], 2)
198
199     while True:
200       drop_ubsan_files(working_dir)
201       # Construct a new evaluation FunC file
202       evfc = gen_eval_fc(working_dir, 3, 4)
203       assert(evfc)
204
205       # Create optimized and non optimized fif-versions of it
206       o2fif = gen_eval_fif(evfc, working_dir, 2)
207       if not o2fif:
208         archive_files(i, working_dir, func_compile_fails)
209         i+=1
210         continue
211
212       o0fif = gen_eval_fif(evfc, working_dir, 0)
213       if not o0fif:
214         archive_files(i, working_dir, func_compile_fails)
215         i+=1
216         continue
217
218       # Run the comparison both ways, and for several different args. If any
output, make sure to archive the files
219       for (l, r) in inputs:
220         ret = run_comparison_both_ways(working_dir, o0fif, o2fif, l, r)
221         if ret:
222           archive_files(i, working_dir, fift_run_fails)
223           i+=1
224
225   if __name__ == "__main__":
```

```
226      main()
```

Figure F.3: Basic Python program used to detect optimization differentials

Verifying results according to a model

The code in Figure F.4 is another approach we used to validate the FunC compiler implementation. For this case, we construct a Python model for FunC expression and then generate arbitrary such expressions. Similar to the optimization differential evaluation case, we use a tiny amount of handwritten Fift code to perform the evaluation. The model computed value is hardcoded into the Fift program, after which the FunC-generated code is run. Finally, the results are compared to ensure they are equal.

A merit of this approach is that the model does not have to be completely accurate as long as it does not produce too many false positives. It can be iteratively refined to be more and more accurate as issues are triaged to be either real findings or model errors.

This approach identified several issues, among them TOB-TON-42, TOB-TON-43, and TOB-TON-47.

```python
import operator
import random
import subprocess

from math import ceil


# Attempts to construct an expression in python of arbitrary large integers and evaluate the corresponding
# FunC counterpart

class Expr:
  def eval(self):
    """Return evaluated result, min, max intermediate values"""
    assert False

  def __repr__(self):
    assert False


class Value(Expr):
  def __init__(self, value = None):
    # 25% chance of large integers
    if value is None:
      self.val = random.randint(-2**256, 2**256) if random.randint(0,4) == 0 else random.randint(-16, 16)
    else:
      self.val = value

  def eval(self):
    return (self.val, self.val, self.val)

  def __repr__(self):
    return f"{self.val}"

class Variable(Value):
  name_idx = 0
  instances = []

  def __init__(self):
    super().__init__()
    self.name = f"var{Variable.name_idx}"
    Variable.name_idx += 1
    Variable.instances.append(self)

  def definition(self):
```

```
45        return f"var {self.name} = {self.eval()[0]};\n"
46
47      @staticmethod
48      def reset():
49        Variable.name_idx = 0
50        Variable.instances = []
51
52      def __repr__(self):
53        return self.name
54
55    class FunctionCall(Value):
56      name_idx = 0
57      instances = []
58
59      def __init__(self):
60        super().__init__()
61        self.name = f"func{FunctionCall.name_idx}"
62        FunctionCall.name_idx += 1
63        FunctionCall.instances.append(self)
64
65      def definition(self):
66        return f"var {self.name}() {{\n\treturn {self.eval()[0]};\n}}\n"
67
68      @staticmethod
69      def reset():
70        FunctionCall.name_idx = 0
71        FunctionCall.instances = []
72
73      def __repr__(self):
74        return f"{self.name}()"
75
76
77    class UnOp(Expr):
78      def __init__(self, opstr, opf, arg):
79        self.opstr = opstr
80        self.opf = opf
81        self.arg = arg
82
83      def eval(self):
84        ares, amin, amax = self.arg.eval()
85        res = self.opf(ares)
86        return (res, min(amin, res), max(amax, res))
87
88      def __repr__(self):
89        return f"({self.opstr} {self.arg})"
90
```

```
 91    class Invert(UnOp):
 92      def __init__(self, arg):
 93        super().__init__('~', operator.invert, arg)
 94
 95    class Negate(UnOp):
 96      def __init__(self, arg):
 97        super().__init__('-', operator.neg, arg)
 98
 99    class BinOp(Expr):
100      def __init__(self, opstr, opf, l, r):
101        self.opstr = opstr
102        self.opf =opf
103        self.l = l
104        self.r = r
105
106      def eval(self):
107        lres, lmin, lmax = self.l.eval()
108        rres, rmin, rmax = self.r.eval()
109        res = self.opf(lres, rres)
110        return (res, min(lmin, rmin, res), max(lmax, rmax, res))
111
112      def __repr__(self):
113        return f"({self.l} {self.opstr} {self.r})"
114
115    class Add(BinOp):
116      def __init__(self, l, r):
117        super().__init__('+', operator.add, l, r)
118
119    class Sub(BinOp):
120      def __init__(self, l, r):
121        super().__init__('-', operator.sub, l, r)
122
123
124    class Mul(BinOp):
125      def __init__(self, l, r):
126        super().__init__('*', operator.mul, l, r)
127
128    class Div(BinOp):
129      def __init__(self, l, r):
130        super().__init__('/', operator.floordiv, l, r)
131
132    class DivRound(BinOp):
133      def __init__(self, l, r):
134        super().__init__('~/', lambda x,y: round(operator.truediv(x, y)), l, r)
135
136    class DivCeil(BinOp):
```

```
137       def __init__(self, l, r):
138         super().__init__('^/', lambda x,y: ceil(operator.truediv(x, y)), l, r)
139
140     class Mod(BinOp):
141       def __init__(self, l, r):
142         super().__init__('%', operator.mod, l, r)
143
144     class BitAnd(BinOp):
145       def __init__(self, l, r):
146         super().__init__('&', operator.and_, l, r)
147
148     class BitOr(BinOp):
149       def __init__(self, l, r):
150         super().__init__('|', operator.or_, l, r)
151
152     class BitXor(BinOp):
153       def __init__(self, l, r):
154         super().__init__('^', operator.xor, l, r)
155
156     class ShiftLeft(BinOp):
157       def __init__(self, l, r):
158         super().__init__('<<', operator.lshift, l, r)
159
160       def eval(self):
161         rres, rmin, rmax = self.r.eval()
162         if rres < 0 or rres > 1023:
163           raise ValueError("ShiftLeft out of range")
164
165         lres, lmin, lmax = self.l.eval()
166         res = self.opf(lres, rres)
167         return (res, min(lmin, rmin, res), max(lmax, rmax, res))
168
169     class ShiftRight(BinOp):
170       def __init__(self, l, r):
171         super().__init__('>>', operator.rshift, l, r)
172
173       def eval(self):
174         rres, rmin, rmax = self.r.eval()
175         if rres < 0 or rres > 1023:
176           raise ValueError("ShiftRight out of range")
177
178         lres, lmin, lmax = self.l.eval()
179         res = self.opf(lres, rres)
180         return (res, min(lmin, rmin, res), max(lmax, rmax, res))
181
182     tvm_true = -1
```

```
183     tvm_false = 0
184     tvm_one = 1
185     tvm_zero = tvm_false
186     tvm_minus_one = tvm_true
187
188     class Rel(BinOp):
189       """Converts into tvm_bools"""
190       def __init__(self, opstr, evalf, l, r):
191         super().__init__(opstr, lambda x, y: tvm_true if evalf(x, y) else
tvm_false, l, r)
192
193     class Eq(Rel):
194       def __init__(self, l, r):
195         super().__init__('==', operator.eq, l, r)
196
197     class Neq(Rel):
198       def __init__(self, l, r):
199         super().__init__('!=', operator.ne, l, r)
200
201     class Lt(Rel):
202       def __init__(self, l, r):
203         super().__init__('<', operator.lt, l, r)
204
205     class Gt(Rel):
206       def __init__(self, l, r):
207         super().__init__('>', operator.gt, l, r)
208
209     class Le(Rel):
210       def __init__(self, l, r):
211         super().__init__('<=', operator.le, l, r)
212
213     class Ge(Rel):
214       def __init__(self, l, r):
215         super().__init__('>=', operator.ge, l, r)
216
217     class IntComp(BinOp):
218       @staticmethod
219       def cmp(l, r):
220         if l < r:
221           return tvm_minus_one
222         elif l > r:
223           return tvm_one
224         else:
225           return tvm_zero
226
227       def __init__(self, l, r):
```

```
228         super().__init__('<=>', IntComp.cmp, l, r)
229
230     def gen_expr():
231
232       ops = [Add, Sub, Mul, Div, BitAnd, BitOr, BitXor, Mod, Invert, Negate,
ShiftLeft, ShiftRight, Eq, Neq, Lt, Gt, Le, Ge, IntComp]
233       n = random.randint(0,2*len(ops))
234       if n < len(ops):
235         op = ops[n]
236         if issubclass(op, BinOp):
237           return op(gen_expr(), gen_expr())
238         elif issubclass(op, UnOp):
239           return op(gen_expr())
240       elif n < len(ops)*1.25:
241         return Variable()
242       elif n < len(ops)*1.75:
243         return FunctionCall()
244       else:
245         return Value()
246
247     while True:
248
249       Variable.reset()
250       FunctionCall.reset()
251
252       e = gen_expr()
253
254       expect_fail =  False
255       bits_needed = 0
256       try:
257         evaluated, min_val, max_val = e.eval()
258         expect_fail = min_val < -2**256 or max_val > 2**256
259       except ZeroDivisionError:
260         expect_fail =  True
261         # Just do a comparison with a dummy value, expecting a overflow anyway
262         evaluated = 777
263       except ValueError:
264         expect_fail = True
265         evaluated = 999
266       except OverflowError:
267         expect_fail = True
268         evaluated = 888
269
270       expr = str(e)
271
272       varlist = "\n".join(["\t" + x.definition() for x in Variable.instances])
```

```
273       functions = "\n".join([x.definition() for x in FunctionCall.instances])
274       program = (
275         f"{functions}\n"
276         f"() evaluate(int l, int r) impure {{\n"
277         f"\tthrow_unless(345, l == r);\n"
278         f"}}\n"
279         f"int main() {{\n"
280         f"{varlist}\n"
281         f"\tevaluate({expr}, {evaluated});\n"
282         f"\treturn 0;\n"
283         f"}}"
284       )
285
286       # print(f"---------\n{program}\n--------")
287       ret = subprocess.run(["./crypto/func", "-I", "-P", "-A"],
input=bytes(program, 'utf-8'), stdout=subprocess.PIPE, stderr=subprocess.PIPE)
288
289       slice_not_cell = ret.stdout.decode('utf-8').replace("END>c", "END>s")
290       vmrun = f"{slice_not_cell}\nrunvmdict .s\n"
291
292       with open('currfift.fif', 'w') as f:
293         f.write(vmrun)
294       ret = subprocess.run(["./crypto/fift", "-I", "../crypto/fift/lib",
"currfift.fif"], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
295
296       successful = ret.returncode == 0 or ret.returncode == 1
297       out = ret.stdout.decode("utf-8")
298       err = ret.stdout.decode("utf-8")
299
300       failed_eq = "0 345" in out
301       failed_overflow = "0 4" in out
302       failed_outofrange = "0 5" in out
303
304
305       did_fail = failed_eq or failed_overflow or failed_outofrange or not
successful
306
307       if did_fail and not expect_fail or expect_fail and not did_fail:
308         print(f"\n========================================\n")
309         print("PROGRAM:\n")
310         print(program)
311         print("FIFT:\n")
312         print(vmrun)
313         print(f"Returncode {ret.returncode}\n")
314         print(f"Out: {out}\n")
315         print(f"Err: {err}\n")
```

```
316         print(f"Expected fail: {expect_fail}\n")
317         print(f"Min value: {min_val}\n")
318         print(f"Max value: {max_val}\n")
319         print(f"Bits needed for intermediate {bits_needed}\n")
320         print(f"Did fail {did_fail}\n\tfailed_eq:
{failed_eq}\n\tfailed_overflow: {failed_overflow}\n\tfail_outofrange:
{failed_outofrange}\n\tnot successful: {not successful}\n")
321         print(f"=========================================\n")
```

Figure F.4: Evaluation of FunC compilation using a Python model

G. Compiler Mitigations

Compiler settings were not audited during the engagement. We recommend reviewing the settings in order to harden production builds as much as possible. The following table lists the basic compiler flags that should be used for hardening.

GCC or Clang Flag	What It Enables or Does
`-z noexecstack`	This flag marks the program's data sections (including the stack and heap) as non-executable (NX). This makes it more difficult for an attacker to execute shellcode. Attackers who wish to bypass NX must resort to return-oriented programming (ROP), an exploitation method that is more difficult as well as less reliable across different builds of a program. This mitigation is enabled by default.
`-Wl,-z,relro,-z,now`	This flag enables full RELRO (relocations read-only). Segments are read-only after relocation, and lazy bindings are disabled. It is a mitigation technique used to harden the data sections of an ELF process. It has three modes of operation: disabled, partial, and full. When a program uses a function from a dynamically loaded library, the function address is stored in the GOT.PLT section (Global Offset Table for Procedure Linkage Table). When RELRO is disabled, each function address entry in the GOT.PLT table points to a dynamic resolver that resolves the entry to the actual address of the intended function when it is first called. In such a case, the memory location of the address is both readable and writable. As a result, an attacker who has control over the process control flow could change the entry of a given function in GOT.PLT to point to any other executable address. For example, the attacker could change the puts function's GOT.PLT entry to point to a system function. Then, if the program called `puts("bin/sh")`, `system("/bin/sh")` would be called instead. When RELRO is fully enabled, the dynamic resolver resolves all of the addresses upon a program's startup and changes the permissions of data

	sections (and therefore `GOT.PLT`) to read-only.

`-fstack-protector-all`
 	
Or (less secure):
 	
`-fstack-protector-strong --param ssp-buffer-size=4`
	This flag adds stack canaries for all functions. Note that this flag may affect the collector's performance.
	
	Stack canaries (stack cookies) make it more difficult to exploit buffer overflow vulnerabilities. A stack canary is a global randomly generated value that is copied to the stack between the stack variables and stack metadata in a function's prologue. When a function returns, the canary on the stack is checked against the global value. The program exits if there is a mismatch, making it more difficult for an attacker to overwrite the return address on the stack. In certain circumstances, attackers may be able to bypass this mitigation by leaking the cookie through a separate information leak vulnerability or by brute-forcing the cookie byte by byte.
	
	To protect only functions that have buffers, use the alternative version indicated.

`-fPIE -pie`	This flag compiles the source as a `PIE`, which `ASLR` depends on.

`-D_FORTIFY_SOURCE=2 -O2`
 	
Or (less secure):
 	
`-D_FORTIFY_SOURCE=1 -O1`
	This flag enables `FORTIFY_SOURCE` protections. These protections require an appropriate optimization flag (`-O1` or `-O2`).
	
	The protection is a glibc-specific feature that enables a series of mitigations primarily aimed at preventing buffer overflows. With a `FORTIFY_SOURCE` level of 1, glibc will add compile-time warnings when potentially unsafe calls to common libc functions (e.g., memcpy and strcpy) are made. With a `FORTIFY_SOURCE` level of 2, glibc will add more stringent runtime checks to these functions and enable a number of lesser-known mitigations. For example, it will disallow the use of the %n format specifier in format strings that are not located in read-only memory pages. This will prevent overwriting data (and gaining code execution) with format string vulnerabilities.
	
	The latter version is less secure, as it enables only compile-time measures; the former adds additional

	runtime checks, which may affect the collector's performance.
`-fstack-clash-protection`	This flag adds checks to functions that may allocate a large amount of memory on the stack to ensure that the new stack pointer and stack frame will not overlap with another memory region, such as the heap. It mitigates a "stack clash vulnerability" in which a program's stack memory region grows so much that it overlaps with another memory region. This bug makes the program confuse the stack memory address with another memory address (e.g., that of the heap); as a result, the regions' data will overlap, which could lead to a denial of service or to control flow hijacking. The stack clash protection mitigation adds explicit memory probing to any function that allocates a large amount of stack memory; when explicit memory probing is used, the function's stack allocation will never make the stack pointer jump over the stack memory guard page, which is located before the stack.

`-fsanitize=cfi`
`-fvisibility=hidden`
`-flto`
 	
(Clang/LLVM only)
	This flag enables CFI checks that help prevent control flow hijacking.

`-fsanitize=safe-stack` *(Clang/LLVM only)*	This flag enables `SafeStack`, which splits the stack frames of certain functions into a safe stack and an unsafe stack, making hijacking of the program's control flow more difficult (Clang/LLVM only).

`-Wall -Wextra -Wpedantic`
`-Wshadow -Wconversion`
`-Wformat-security`
	This flag enables compile-time checks and warnings.

System	What It Enables or Does
`ASLR (Address Space Layout Randomization)`	This feature randomizes the memory location of each section of the program. This makes it more difficult for an attacker to write reliable exploits, primarily by impeding jumps to ROP gadgets. ASLR requires

	cooperation from both the system and the compiler. To fully support ASLR, a program must be compiled as a position-independent executable (PIE). Most of the Linux distributions have ASLR enabled. This can be checked by reading the value stored in the `/proc/sys/kernel/randomize_va_space` file: 0 means that ASLR is disabled, 1 means it is partially enabled (only some bits of the addresses are randomized), and 2 means it is fully enabled. This file is writable, and an admin can disable or enable the mitigation. An information leak in the program may enable an attacker to bypass ASLR.

H. Opcode Timing and Gas Analysis

We implemented a utility to compare the timing of VM execution against the gas used. The goal was to discover opcodes or opcode sequences that consume an inordinate amount of computational resources relative to their gas cost. Its source code is listed in Figure H.1.

The utility expects two command line arguments, each a hex string: The TVM code used to set up the stack and VM state followed by the TVM code to measure. For example, to test the `DIVMODC` opcode:

```
$ test-timing 80FF801C A90E 2>/dev/null
OPCODE,runtime mean,runtime stddev,gas mean,gas stddev
A90E,0.0066416,0.00233496,26,0
```

The runtime is listed in milliseconds.

```
#include <ctime>
#include <iomanip>

#include "vm/vm.h"
#include "vm/cp0.h"
#include "vm/dict.h"
#include "fift/utils.h"
#include "common/bigint.hpp"

#include "td/utils/base64.h"
#include "td/utils/tests.h"
#include "td/utils/ScopeGuard.h"
#include "td/utils/StringBuilder.h"

td::Ref<vm::Cell> to_cell(const unsigned char *buff, int bits) {
  return vm::CellBuilder().store_bits(buff, bits, 0).finalize();
}

long double timingBaseline;

typedef struct {
  long double mean;
  long double stddev;
} stats;

struct runInfo {
  long double runtime;
  long long gasUsage;
  int vmReturnCode;

  runInfo() : runtime(0.0), gasUsage(0), vmReturnCode(0) {}
  runInfo(long double runtime, long long gasUsage, int vmReturnCode) :
      runtime(runtime), gasUsage(gasUsage), vmReturnCode(vmReturnCode) {}

  runInfo operator+(const runInfo& addend) const {
    return {runtime + addend.runtime, gasUsage + addend.gasUsage, vmReturnCode ? vmReturnCode :
addend.vmReturnCode};
```

```
  }

  runInfo& operator+=(const runInfo& addend) {
    runtime += addend.runtime;
    gasUsage += addend.gasUsage;
    if(!vmReturnCode && addend.vmReturnCode) {
      vmReturnCode = addend.vmReturnCode;
    }
    return *this;
  }

  bool errored() const {
    return vmReturnCode != 0;
  }
};

typedef struct {
  stats runtime;
  stats gasUsage;
  bool errored;
} runtimeStats;

runInfo time_run_vm(td::Slice command) {
  unsigned char buff[128];
  const int bits = (int)td::bitstring::parse_bitstring_hex_literal(buff, sizeof(buff),
command.begin(), command.end());
  CHECK(bits >= 0);

  const auto cell = to_cell(buff, bits);

  vm::init_op_cp0();
  vm::DictionaryBase::get_empty_dictionary();

  class Logger : public td::LogInterface {
   public:
    void append(td::CSlice slice) override {
      res.append(slice.data(), slice.size());
    }
    std::string res;
  };
  static Logger logger;
  logger.res = "";
  td::set_log_fatal_error_callback([](td::CSlice message) {
td::default_log_interface->append(logger.res); });
  vm::VmLog log{&logger, td::LogOptions::plain()};
  log.log_options.level = 4;
  log.log_options.fix_newlines = true;
  log.log_mask |= vm::VmLog::DumpStack;

  vm::Stack stack;
  try {
    vm::GasLimits gas_limit(10000, 10000);

    std::clock_t cStart = std::clock();
    int ret = vm::run_vm_code(vm::load_cell_slice_ref(cell), stack, 0 /*flags*/, nullptr
/*data*/,
                              std::move(log) /*VmLog*/, nullptr, &gas_limit);
    std::clock_t cEnd = std::clock();
    const auto time = (1000.0 * static_cast<long double>(cEnd - cStart) / CLOCKS_PER_SEC) -
timingBaseline;
```

```
    return {time >= 0 ? time : 0, gas_limit.gas_consumed(), ret};
  } catch (...) {
    LOG(FATAL) << "catch unhandled exception";
    return {-1, -1, 1};
  }
}

runtimeStats averageRuntime(td::Slice command) {
  const size_t samples = 5000;
  runInfo total;
  std::vector<runInfo> values;
  values.reserve(samples);
  for(size_t i=0; i<samples; ++i) {
    const auto value = time_run_vm(command);
    values.push_back(value);
    total += value;
  }
  const auto runtimeMean = total.runtime / static_cast<long double>(samples);
  const auto gasMean = static_cast<long double>(total.gasUsage) / static_cast<long
double>(samples);
  long double runtimeDiffSum = 0.0;
  long double gasDiffSum = 0.0;
  bool errored = false;
  for(const auto value : values) {
    const auto runtime = value.runtime - runtimeMean;
    const auto gasUsage = static_cast<long double>(value.gasUsage) - gasMean;
    runtimeDiffSum += runtime * runtime;
    gasDiffSum += gasUsage * gasUsage;
    errored = errored || value.errored();
  }
  return {
      {runtimeMean, sqrt(runtimeDiffSum / static_cast<long double>(samples))},
      {gasMean, sqrt(gasDiffSum / static_cast<long double>(samples))},
      errored
  };
}

runtimeStats timeInstruction(const std::string& setupCode, const std::string& toMeasure) {
  const auto setupCodeTime = averageRuntime(setupCode);
  const auto totalCodeTime = averageRuntime(setupCode + toMeasure);
  return {
    {totalCodeTime.runtime.mean - setupCodeTime.runtime.mean, totalCodeTime.runtime.stddev},
    {totalCodeTime.gasUsage.mean - setupCodeTime.gasUsage.mean, totalCodeTime.gasUsage.stddev}
  };
}

int main(int argc, char** argv) {
  if(argc != 2 && argc != 3) {
    std::cerr << "Usage: " << argv[0] <<
        " [TVM_SETUP_BYTECODE_HEX] TVM_BYTECODE_HEX" << std::endl << std::endl;
    return 1;
  }
  std::cout << "OPCODE,runtime mean,runtime stddev,gas mean,gas stddev" << std::endl;
  timingBaseline = averageRuntime("").runtime.mean;
  std::string setup, code;
  if(argc == 2) {
    setup = "";
    code = argv[1];
  } else {
    setup = argv[1];
```

```
    code = argv[2];
  }
  const auto time = timeInstruction(setup, code);
  std::cout << code << "," << time.runtime.mean << "," << time.runtime.stddev << "," <<
      time.gasUsage.mean << "," << time.gasUsage.stddev << std::endl;
  return 0;
}
```

Figure H.1: Utility for timing opcodes

I. Method ID Collisions

TON requires each procedure in a contract to have a unique method ID. FunC uses a CRC16 checksum to auto-generate these method IDs. Specifically, FunC uses

`(crc16(`*procedure name*`) & 0xffff) | 0x10000`

as the auto-generated method ID.

Trail of Bits developed a script to automatically generate procedure names that collide with—i.e., produce the same method ID as—a given procedure. There is no single, standard CRC16 specification. The algorithm that TON uses is given in FIgure I.1.

```
 1    def crc16(data):
 2        """CRC 16 implementation from TON
 3
 4
https://github.com/ton-blockchain/ton/blob/d11580dfb3b81ea5d00775502737d59c155adfb2/
tdutils/td/utils/crypto.cpp#L1177-L1205
 5
 6        """
 7        crc = 0
 8        for c in data:
 9            t = (c ^ (crc >> 8)) & 0xFF
10            crc = crc16_table[t] ^ (crc << 8)
11        return crc
```

Figure I.1: Python implementation of FunC's crc16 implementation

We reproduce this implementation in Satisfiability Modulo Theories (SMT), and then ask a SMT solver to enumerate all possible values of `data` from line 1 of Figure I.1 that would produce a `crc` value on line 11 that would collide with the given procedure name's CRC16. This approach is capable of almost instantaneously yielding collisions for all inputs using the Z3 theorem prover.

Our code for enumerating these collisions is given in Figure I.2.

```
 1    from typing import Iterable
 2
 3    import z3
 4
 5
 6    crc16_table = [
 7        0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7, 0x8108,
0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad,
 8        0xe1ce, 0xf1ef, 0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7,
0x62d6, 0x9339, 0x8318, 0xb37b, 0xa35a,
 9        0xd3bd, 0xc39c, 0xf3ff, 0xe3de, 0x2462, 0x3443, 0x0420, 0x1401, 0x64e6,
```

```
0x74c7, 0x44a4, 0x5485, 0xa56a, 0xb54b,
  10        0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d, 0x3653, 0x2672, 0x1611,
0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
  11        0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc, 0x48c4,
0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861,
  12        0x2802, 0x3823, 0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a,
0xb92b, 0x5af5, 0x4ad4, 0x7ab7, 0x6a96,
  13        0x1a71, 0x0a50, 0x3a33, 0x2a12, 0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e, 0x9b79,
0x8b58, 0xbb3b, 0xab1a, 0x6ca6, 0x7c87,
  14        0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41, 0xedae, 0xfd8f, 0xcdec,
0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49,
  15        0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70, 0xff9f,
0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a,
  16        0x9f59, 0x8f78, 0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e,
0xe16f, 0x1080, 0x00a1, 0x30c2, 0x20e3,
  17        0x5004, 0x4025, 0x7046, 0x6067, 0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d,
0xd31c, 0xe37f, 0xf35e, 0x02b1, 0x1290,
  18        0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256, 0xb5ea, 0xa5cb, 0x95a8,
0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d,
  19        0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405, 0xa7db,
0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e,
  20        0xc71d, 0xd73c, 0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615,
0x5634, 0xd94c, 0xc96d, 0xf90e, 0xe92f,
  21        0x99c8, 0x89e9, 0xb98a, 0xa9ab, 0x5844, 0x4865, 0x7806, 0x6827, 0x18c0,
0x08e1, 0x3882, 0x28a3, 0xcb7d, 0xdb5c,
  22        0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a, 0x4a75, 0x5a54, 0x6a37,
0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
  23        0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9, 0x7c26,
0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83,
  24        0x1ce0, 0x0cc1, 0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9,
0x9ff8, 0x6e17, 0x7e36, 0x4e55, 0x5e74,
  25        0x2e93, 0x3eb2, 0x0ed1, 0x1ef0]
  26
  27
  28    def crc16_table_lookup(t):
  29        if isinstance(t, int):
  30            return crc16_table[t]
  31        ret = z3.BitVecVal(crc16_table[0], 32)
  32        for i, value in enumerate(crc16_table[1:]):
  33            ret = z3.If(t == i + 1, z3.BitVecVal(value, 32), ret)
  34        return ret
  35
  36
  37    def crc16(data, start_crc=0):
  38        """CRC 16 implementation from TON
  39
  40
https://github.com/ton-blockchain/ton/blob/d11580dfb3b81ea5d00775502737d59c155adfb2/
tdutils/td/utils/crypto.cpp#L1177-L1205
  41
  42       """
  43        crc = start_crc
  44        for c in data:
```

```
  45            t = (c ^ (crc >> 8)) & 0xFF
  46            crc = crc16_table_lookup(t) ^ (crc << 8)
  47        return crc
  48
  49
  50    def build_problem(solver, string_length: int, with_prefix: str = "",
prev_hash = None, _char_vars = None):
  51        if _char_vars is None:
  52            _char_vars = []
  53        if prev_hash is None:
  54            prev_hash = z3.BitVec("init_hash", 32)
  55            solver.add(prev_hash == 0)
  56        if string_length <= 0:
  57            return prev_hash, _char_vars
  58        else:
  59            c = z3.BitVec("c" + str(string_length), 32)
  60
  61            if with_prefix:
  62                solver.add(c == ord(with_prefix[0]))
  63            else:
  64                # # The following two constraints allow function names with
basically any ASCII characters:
  65                # solver.add(c >= 33)
  66                # solver.add(c <= 122)
  67
  68                # # The following constraint only permits function names with
mixed case characters:
  69                # solver.add(z3.Or(
  70                #     z3.And(c >= ord('A'), c <= ord('Z')),
  71                #     z3.And(c >= ord('a'), c <= ord('z'))
  72                # ))
  73
  74                # # The following two constraints only allow function names with
lower-case characters:
  75                # solver.add(c >= ord('a'))
  76                # solver.add(c <= ord('z'))
  77
  78                # Lower-case, but also allow interior `_`
  79                if string_length > 1 and not _char_vars:
  80                    solver.add(z3.Or(
  81                        z3.And(c >= ord('a'), c <= ord('z')),
  82                        c == ord('_')
  83                    ))
  84                else:
  85                    solver.add(c >= ord('a'))
  86                    solver.add(c <= ord('z'))
  87
  88            _char_vars.append(c)
  89            h = z3.BitVec("hash" + str(string_length), 32)
  90            next_crc = crc16((c,), start_crc=prev_hash)
  91            solver.add(h == next_crc)
  92            return build_problem(solver, string_length=string_length - 1,
with_prefix=with_prefix[1:], prev_hash=h,
```

```
  93                                  _char_vars=_char_vars)
  94
  95
  96      def solve(hash_to_collide: int, starting_length: int = 1, enumerate_all:
bool = True, with_prefix: str = ""):
  97          i = max(starting_length, len(with_prefix))
  98          yielded = False
  99          while enumerate_all or not yielded:
 100              print(f"Trying string length {i}...")
 101
 102              solver = z3.Solver()
 103
 104              h, char_vars = build_problem(solver, i, with_prefix=with_prefix)
 105              solver.add((h & 0xffff) | 0x10000 == hash_to_collide)
 106
 107              while solver.check() == z3.sat:
 108                  m = solver.model()
 109                  yield bytes(m[c].as_long() for c in char_vars)
 110                  yielded = True
 111                  asmts = []
 112                  for c in char_vars:
 113                      asmts.append(c != m[c])
 114                  solver.add(z3.Or(*asmts))  # prevent next model from using the
same assignment as a previous model
 115
 116              i += 1
 117
 118
 119      def dict_collisions(to_match: Iterable[str] = (), with_prefix: str = ""):
 120          method_ids = dict()
 121          to_match_ids = {
 122              fname: (crc16(fname.encode("utf-8")) & 0xffff) | 0x10000 for fname
in to_match
 123          }
 124          with open("/usr/share/dict/words", "r") as f:
 125              for line in f:
 126                  line = f"{with_prefix}{line.strip()}"
 127                  if len(line) < 3:
 128                      continue
 129                  method_id = (crc16(line.encode("utf-8")) & 0xffff) | 0x10000
 130                  if to_match_ids and method_id not in to_match_ids:
 131                      continue
 132                  elif method_id not in method_ids:
 133                      method_ids[method_id] = {line}
 134                  else:
 135                      method_ids[method_id].add(line)
 136          for method_id, funcnames in method_ids.items():
 137              if not to_match_ids and len(funcnames) < 2:
 138                  continue
 139              print(f"{method_id}\t{','.join(funcnames)}")
 140
 141
 142      if __name__ == "__main__":
```

```
143         import sys
144
145         if len(sys.argv) >= 3 and sys.argv[1] == "--with-prefix":
146             prefix = sys.argv[2]
147             args = sys.argv[2:]
148         else:
149             prefix = ""
150             args = sys.argv
151
152         if len(args) >= 2 and args[1] == "--dict-collisions":
153             dict_collisions(args[2:], with_prefix=prefix)
154             exit(0)
155         elif len(args) > 1:
156             funcnames = (f.encode("utf-8") for f in args[1:])
157         else:
158             funcnames = (b"main",)
159
160         for to_collide in funcnames:
161             crc = crc16(to_collide)
162             method_id = (crc & 0xffff) | 0x10000
163             print(f"Finding a hash collision for method_id =
((crc16({to_collide!r}) = {crc}) & 0xffff) | 0x10000 "
164                   f"= {method_id}")
165             for collision in solve(method_id, with_prefix=prefix):
166                 if collision == to_collide:
167                     continue
168                 collision_crc = crc16(collision)
169                 collision_method_id = (collision_crc & 0xffff) | 0x10000
170                 assert method_id == collision_method_id
171                 print(f"Collision: {collision.decode('utf-8')!r}
(crc16={collision_crc}, method_id={collision_method_id})")
172                 sys.stdout.flush()
```

Figure I.2: Trail of Bits's script to enumerate FunC method ID collisions

J. 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 March 27 to March 31, 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.

TON's fixes span a number of commits, branches, repositories, and forks. We list the associated location of each fix in the following Detailed Fix Review Results section. Note that not all of the fixes have yet to be merged into the TON master branch or deployed to the mainnet. There is only one latent high-severity finding that has yet to be resolved, TOB-TON-36; this is because at least one commercial product (a DEX) would break due to the fix. TON reports that it is working with the DEX to upgrade the contracts before deploying a network-wide fix.

In summary, of the 54 issues described in this report, TON has resolved 37, has partially resolved 3, and has not resolved 11; additionally, Trail of Bits redacted three issues of previously undetermined severity. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Proxied ADNL pong messages may have empty data	Informational	Resolved
2	A block ID with no associated queue will cause a crash	Informational	Resolved
3	Token manager only checks every other download for timeouts	High	Resolved
4	FunC compiler will dereference an invalid pointer when output file is provided	Low	Resolved
5	ListIterator postfix increment operator returns a local variable by reference	Undetermined	Resolved

6	TVM programs can trigger undefined behavior in bigint.hpp	High	Resolved
7	TVM programs can trigger undefined behavior in bitstring.cpp	High	Resolved
8	TVM programs can trigger undefined behavior in tonops.cpp	High	Resolved
9	TVM programs can trigger undefined behavior in CellBuilder.cpp	High	Resolved
10	Multiple Fift stack instructions fail to check the stack depth	Low	Resolved
11	PUSHPOW2 opcode uses twice as much CPU time as opcodes with a similar gas cost	Low	Unresolved
12	Stack use-after-scope in tdutils test	Informational	Resolved
13	On-chain pseudorandom number generation	Informational	Partially resolved
14	The NOW opcode can cause consensus issues	Undetermined	Retracted
15	VM state guards fail when not assigned to a variable	Low	Resolved
16	Performance warning timers in the cell DB do not work	Low	Resolved
17	DHT queries will crash if debug logging is enabled	Low	Resolved
18	Frequent connection state changes can cause an ADNL node to exhaust memory	Informational	Resolved
19	Missing base copy constructor invocation in derived copy constructor	Informational	Unresolved

20	Unbounded storage of received Catchain blocks	Informational	Resolved
21	Getting account state can crash when building a state root proof	High	Resolved
22	Misaligned object allocation and interaction	High	Resolved
23	Use of DowncastHelper leads to invalid downcast of incorrect type	High	Resolved
24	Clock drift can break consensus	Informational	Resolved
25	Shard records can be instantiated with uninitialized member variables	Undetermined	Resolved
26	Signatures of block antecessors are not validated	Undetermined	Unresolved
27	TLB reference validation can be bypassed	Undetermined	Resolved
28	The TON client's get shards request can fail	Low	Resolved
29	Bigint and cell tests can silently fail due to undefined behavior	Low	Partially resolved
30	Multiplication of a constant can lead to a misaligned stack	High	Resolved
31	FunC codegen invokes undefined behavior	Medium	Resolved
32	Constant operations on NaN can cause the FunC compiler to crash	Low	Resolved
33	Undefined variables in FunC are treated as undefined functions and do not cause a compiler error	Medium	Unresolved

34	Calls to implicitly impure functions without a return value are always optimized out without an error	Medium	Unresolved
35	Calls to implicitly impure functions with unused return values are always optimized out without an error	Informational	Unresolved
36	Comparison to NaN results in the other comparand	High	Unresolved
37	FunC fails to reject out-of-range constants	Low	Resolved
38	Inconsistent runtime behavior for operations resulting in NaN	Medium	Resolved
39	Missing _Bit-marker for positive integer 1	Informational	Resolved
40	Method IDs can collide without warning	Low	Unresolved
41	Single-line comments are honored within multi-line comments	Low	Resolved
42	Bitwise operators can cause the FunC compiler to crash	Low	Resolved
43	FunC compiler can produce undefined opcodes	Low	Resolved
44	Invalid syntax can cause the FunC compiler to crash	Low	Resolved
45	Dictionary lookup can return incorrect results	High	Partially Resolved
46	Dictionary insertion can inconsistently crash	High	Resolved
47	Bitwise negation of false is not always true	High	Resolved

48	Setting the random number seed from the FunC standard library causes a stack misalignment	Medium	Resolved
49	Querying a dictionary throws exception	Low	Unresolved
50	Compile time integer literal operations can result in unexpected control flow	Low	Resolved
51	Generating a random number throws an exception	Undetermined	Retracted
52	Ethereum bridge signature verification will always pass for address zero	Informational	Unresolved
53	Context sensitivity of the ; token can lead to confusion and bugs	Informational	Unresolved
54	Sign-confusion can lead to votes being collected incorrectly	Undetermined	Retracted

Detailed Fix Review Results

TOB-TON-1: Proxied ADNL pong messages may have empty data
Resolved in commit 34c1c548c45dd86ab1e180d8d154cbd6d7db42ea. The superfluous assignment to `p.data` was removed. Consider changing the name of the local variable `data`, as it shadows a function argument.

TOB-TON-2: A block ID with no associated queue will cause a crash
Resolved in commit 0578cb4a4285cf16e613129b85da21729fab7453. The function will now exit with an error if there is no associated queue.

TOB-TON-3: Token manager only checks every other download for timeouts
Resolved in commit 0578cb4a4285cf16e613129b85da21729fab7453. The iterator is now updated once for every iteration, either as a result of erasing a timed-out item or by advancing it to the next available item.

TOB-TON-4: FunC compiler will dereference an invalid pointer when output file is provided
Resolved in commit 0578cb4a4285cf16e613129b85da21729fab7453. The FunC compiler now uses static constants from `std::fstream`.

TOB-TON-5: ListIterator postfix increment operator returns a local variable by reference
Resolved in commit 0578cb4a4285cf16e613129b85da21729fab7453. The postfix increment operator now returns the local variable by value instead of by reference.

TOB-TON-6: TVM programs can trigger undefined behavior in bigint.hpp reference
Resolved in commit 9c6787d2ff27dcb29fad562a9ca64ac650d66d34. Using the byte-code sequences described in the issue, we are no longer able to reach undefined behavior. The TON team has updated the affected operations: most of them now cast to unsigned types, and a few of them have been replaced with different operations.

In our verification of this fix, we determined only that the operations can no longer reach the specific undefined behavior we reported. We did not verify that the results of the computations are correct. The fix for line 1925 performs an explicit cast to an unsigned long long integer type instead of using the trait-provided typedef `uword_t`, which could cause errors if these types are not in sync.

TOB-TON-7: TVM programs can trigger undefined behavior in bitstring.cpp
Resolved in commit 9c6787d2ff27dcb29fad562a9ca64ac650d66d34. The TON team changed the affected operations to use unsigned types and implemented additional checks to prevent undefined behavior.

TOB-TON-8: TVM programs can trigger undefined behavior in tonops.cpp

Resolved in commit 9c6787d2ff27dcb29fad562a9ca64ac650d66d34. The TON team changed the affected operations to cast to unsigned types, preventing signed integer overflow.

TOB-TON-9: TVM programs can trigger undefined behavior in CellBuilder.cpp
Resolved in commit 9c6787d2ff27dcb29fad562a9ca64ac650d66d34. The TON team implemented an explicit check for the error case to prevent left-shifting by too many positions.

TOB-TON-10: Multiple Fift stack instructions fail to check the stack depth
Resolved in commit 0578cb4a4285cf16e613129b85da21729fab7453. The instructions now fail with a stack underflow check instead of crashing.

TOB-TON-11: PUSHPOW2 opcode uses twice as much CPU time as opcodes with a similar gas cost
Unresolved as of commit e37583e5e6e8cd0aebf5142ef7d8db282f10692b. The TVM continues to use the same gas cost for all opcodes.

The client provided the following context for this finding's fix status:

> *While indeed, PUSHPOW2 spends more time per gas than some other opcodes it lays within acceptable range. Some other opcodes which spends [sic] unproportionally more time pre gas unit were discovered, thanks to [the benchmark provided by Trail of Bits], and will be fixed during TVM update.*

TOB-TON-12: Stack use-after-scope in tdutils test
Resolved in commit 0578cb4a4285cf16e613129b85da21729fab7453. The TON team re-ordered the local variables to prevent `id` from being accessed after destruction.

TOB-TON-13: On-chain pseudorandom number generation
Partially resolved. Although the TON team has documented the risks of the use of pseudorandom numbers, this warning comes only in the final section of the documentation. We recommend presenting that information upfront to ensure readers do not miss it. Additionally, the added documentation contains a broken link to a reference, in the sentence, "An evil validator with some probability can affect the seed...".

TOB-TON-14: <Retracted> The NOW opcode can cause consensus issues
This issue was originally reported as a finding of undetermined severity, under the incorrect assumption that the `NOW` opcode had the potential to return different values across validators. During the fix review, we discovered that the time is retrieved from the block, which should be consistent across validators, so we have retracted this finding. However, since this value is manipulable by the block collator, we still recommend exercising caution in using it in contracts.

TOB-TON-15: VM state guards fail when not assigned to a variable
Resolved in the SpyCheese fork. The fork extends the lifetime of the `Guard` object by giving it an identifier. This commit has not yet been merged to the TON repository.

TOB-TON-16: Performance warning timers in the cell DB do not work
Resolved in commit `0578cb4a4285cf16e613129b85da21729fab7453`. The `PerfWarningTimer` class was assigned an identifier, so it now correctly measures the execution time of subsequent operations in the function.

TOB-TON-17: DHT queries will crash if debug logging is enabled
Resolved in commit `0578cb4a4285cf16e613129b85da21729fab7453`. The vulnerability is resolved; however, we recommend not relying on invoking virtual functions during object construction, as doing so is prone to error.

TOB-TON-18: Frequent connection state changes can cause an ADNL node to exhaust memory
Resolved in commit `0578cb4a4285cf16e613129b85da21729fab7453`. The `pending_messages_` vector is now explicitly clear, preventing memory from being exhausted.

TOB-TON-19: Missing base copy constructor invocation in derived copy constructor
Unresolved as of commit `e37583e5e6e8cd0aebf5142ef7d8db282f10692b`. This informational-severity finding included a recommendation to make the codebase more robust against future changes. The client provided the following context for this finding's fix status:

> *We believe that since countable objects are unlikely to be changed, this issue can be neglected*

TOB-TON-20: Unbounded storage of received Catchain blocks
Resolved. Although the TON team made no changes to the codebase to fix this issue, the vulnerability is documented and a solution exists, so we consider the issue resolved. We still recommend adding warnings issued to users when the parameter is set to zero..

TOB-TON-21: Getting account state can crash when building a state root proof
Resolved in commit `0578cb4a4285cf16e613129b85da21729fab7453`. The comparison was changed to use `CellHash` instances with valid lifetimes.

TOB-TON-22: Misaligned object allocation and interaction
Resolved in commit `9c6787d2ff27dcb29fad562a9ca64ac650d66d34`. The `alloc` function now correctly accounts for alignment; however, invocations of `alloc` still use a hard-coded alignment that is not automatically derived from the type passed to it. Although the hard-coded value of 8 is likely sufficient for most architectures, we recommend implementing an alignment based on type.

TOB-TON-23: Use of DowncastHelper leads to invalid downcast of incorrect type
Resolved in commit `0578cb4a4285cf16e613129b85da21729fab7453`. The TON team has mitigated the risk of undefined behavior resulting from casting across type hierarchies by implementing and using `downcast_construct` to create objects of a specific type.

TOB-TON-24: Clock drift can break consensus
Resolved in commit `3e92ab9da849feda58c26bab1c25dacc1b7babe7` in the TON community repository. A sentence about the importance of correct time on the node server has been added to the documentation.

TOB-TON-25: Shard records can be instantiated with uninitialized member variables
Resolved in commit `9c6787d2ff27dcb29fad562a9ca64ac650d66d34`. All struct members are now initialized in the default constructor.

TOB-TON-26: Signatures of block antecessors are not validated
Unresolved as of commit `e37583e5e6e8cd0aebf5142ef7d8db282f10692b`.

The client provided the following context for this finding's fix status:

> *It's ok, currently the list of signatures is not included in the block, because if it were, it would require an extra round of approval (all validators need the same set of signatures before moving on to the next block), and this would slow down the consensus.*

We have not observed any protections that would *prevent* a block from containing `prev_blk_signatures`. A malicious collator embedding invalid `prev_blk_signatures` could do so without being caught by the current validator code, resulting in an invalid block on-chain. If signature validation of block antecessors is enabled in the future, old blocks may not be validated and a fork may be created.

TOB-TON-27: TLB reference validation can be bypassed
Resolved in commit `9c6787d2ff27dcb29fad562a9ca64ac650d66d34`. A check was added to ensure that the `ops` argument is positive before it is decremented.

TOB-TON-28: The TON client's get shards request can fail
Resolved in commit `9c6787d2ff27dcb29fad562a9ca64ac650d66d34`. The commented-out return statements were restored.

TOB-TON-29: Bigint and cell tests can silently fail due to undefined behavior
Partially resolved in commit `0578cb4a4285cf16e613129b85da21729fab7453`. The TON team resolved all of the instances of integer overflowing negative bit shifts causing undefined behavior, except for one instance of signed integer overflow in `test-cells.cpp`, which still persists.

TOB-TON-30: Multiplication of a constant can lead to a misaligned stack

Resolved as of commit 701fc6afad4484d6f8df3500ad85123c2de51b2e. The example FunC code now correctly compiles without producing a misaligned stack.

TOB-TON-31: FunC codegen invokes undefined behavior
Resolved in commit 0578cb4a4285cf16e613129b85da21729fab7453. The `next` pointer is now dereferenced only when not `null`.

TOB-TON-32: Constant operations on NaN can cause the FunC compiler to crash
Resolved as of commit 1662cb1bdcf8b7103ab909e373fbfeea5bd61cad, currently merged into the testnet branch. The compiler no longer crashes on constant operations involving NaNs.

TOB-TON-33: Undefined variables in FunC are treated as undefined functions and do not cause a compiler error
Unresolved as of commit e37583e5e6e8cd0aebf5142ef7d8db282f10692b. The compiler continues to permit calls to undefined functions.

The client provided the following context for this finding's fix status:

> *Sometimes FunC is used to generate templates that when [sic] will be used as pieces of larger smartcontracts, so [compiling FunC code with undefined function symbols] is ok. [The] Fift compilation step will cause an error for incomplete code.*

Given the knowledge that FunC is used to generate "templates" with potentially undefined function symbols (similar to C object files), we still recommend having it emit a compiler warning or error on undefined symbols as the default. Additionally, we recommend adding a command line option to `func` equivalent to the C compiler option `` `-c` `` to suppress the warnings/errors.

TOB-TON-34: Calls to implicitly impure functions without a return value are always optimized out without an error
Unresolved as of commit e37583e5e6e8cd0aebf5142ef7d8db282f10692b. The compiler will not issue a warning when eliding implicitly impure functions without a return value.

The client provided the following context for this finding's fix status:

> *It is documented behavior.*

TOB-TON-35: Calls to implicitly impure functions with unused return values are always optimized out without an error
Unresolved as of commit e37583e5e6e8cd0aebf5142ef7d8db282f10692b. The compiler will not issue a warning when eliding implicitly impure functions with unused return values.

The client provided the following context for this finding's fix status:

It is documented behavior.

TOB-TON-36: Comparison to NaN results in the other comparand
Unresolved as of commit e37583e5e6e8cd0aebf5142ef7d8db282f10692b.

The client provided the following context for this finding's fix status:

> *It is indeed quite a severe issue. Unfortunately, as our transactions analysis shows, on mainnet there is at least one commercial project (DEX) which relies on that behavior. In particular, if TVM will be fixed, some swaps will be thrown and funds lost. We are helping this DEX to migrate to correct behavior and will fix TVM after that.*

TOB-TON-37: FunC fails to reject out-of-range constants
Resolved in commit 0578cb4a4285cf16e613129b85da21729fab7453. The compiler now emits an error on out-of-range integer constants.

TOB-TON-38: Inconsistent runtime behavior for operations resulting in NaN
Resolved in commit 0578cb4a4285cf16e613129b85da21729fab7453. A compilation error is now thrown on division-by-zero errors.

TOB-TON-39: Missing _Bit-marker for positive integer 1
Resolved in commit 0578cb4a4285cf16e613129b85da21729fab7453. TON chose to address the problem by removing the erroneous branch and its associated optimizations.

TOB-TON-40: Method IDs can collide without warning
Unresolved as of commit e37583e5e6e8cd0aebf5142ef7d8db282f10692b. The compiler continues to have the ability to emit methods with duplicate IDs that would cause a runtime error when the contract is deployed.

The client provided the following context for this finding's fix status:

> *For now, since Fift throws an error during `fiftasm` compilation, it is considered a minor issue. It is planned to be fixed in the future.*

TOB-TON-41: Single-line comments are honored within multi-line comments
Resolved in commit 34669a4b70e50da253c1ce9974e1da47b71a59bb in the documentation repository. Single-line comments are still honored within multi-line comments, but this behavior is now documented.

TOB-TON-42: Bitwise operators can cause the FunC compiler to crash
Resolved in commit c6143715cc29ae23dad202b2580083099d8f61d2. The compiler no longer crashes when bitwise operators are used on constants.

TOB-TON-43: FunC compiler can produce undefined opcodes

Resolved in commit 0578cb4a4285cf16e613129b85da21729fab7453. The compiler no longer emits the erroneous NEGPOW2 opcode.

TOB-TON-44: Invalid syntax can cause the FunC compiler to crash
Resolved in commit c6143715cc29ae23dad202b2580083099d8f61d2. The compiler now emits an error on invalid syntax.

TOB-TON-45: Dictionary lookup can return incorrect results
Partially resolved in commit 1538e324ee5f3ad339c5a7b06debdeb19414aaaa in the documentation repository. The compiler continues to emit code that behaves unintuitively when different bit-lengths are used on the same dictionary with no warning, but this behavior is now documented.

TOB-TON-46: Dictionary insertion can inconsistently crash
Resolved in commit 1538e324ee5f3ad339c5a7b06debdeb19414aaaa in the documentation repository. The compiler continues to inconsistently emit code that will produce a runtime error when different bit-lengths are used on the same dictionary, but this behavior is now documented.

TOB-TON-47: Bitwise negation of false is not always true
Resolved in commit 91580e7ebf4bcd589581250ce509f51fdd58a66d. The FunC compiler no longer emits incorrect code for the bitwise negation operator.

TOB-TON-48: Setting the random number seed from the FunC standard library causes a stack misalignment
Resolved in commit 3d9a16558679ca48ef5616c3518a3c6fd72a0220. The type signature of the `set_seed` function in the standard library was updated to match its associated opcode, preventing the stack misalignment.

TOB-TON-49: Querying a dictionary throws exception
Unresolved as of commit e37583e5e6e8cd0aebf5142ef7d8db282f10692b in TON and 3171ec4442ac6d214f2993287630a4dbeb04f758 in the documentation repository. Our example FunC code still throws an exception when querying a dictionary for a missing key.

The client provided the following context for this finding's fix status:

> *Behavior documented.*

However, the latest documentation for the `udict_get` function states the following:

> *On success, [`udict_get?`] returns the value found as a slice along with a -1 flag indicating success. If fails, it returns (null, 0).*

This is not the behavior we observe in the latest version of TON; we still get the following exception:

```
exception code 10: invalid dictionary fork node
```

TOB-TON-50: Compile time integer literal operations can result in unexpected control flow
Resolved in commit `0578cb4a4285cf16e613129b85da21729fab7453`. The compiler no longer emits a NaN when literals are out of bounds; instead, it throws a compilation error.

TOB-TON-51: <Retracted> Generating a random number throws an exception
This issue was originally reported as a finding of undetermined severity. In our experiments, the `RANDU256` opcode would throw an exception when run. During the fix review, we discovered that this was because we were not correctly initializing our block parameters, so we have retracted this finding. However, we would like to reiterate our recommendations for finding TOB-TON-13 to discourage the use of on-chain random number generation.

TOB-TON-52: Ethereum bridge signature verification will always pass for address zero
Unresolved as of commit `e37583e5e6e8cd0aebf5142ef7d8db282f10692b`. All signatures from the zero address are still accepted. This behavior is not documented.

The client provided the following context for this finding's fix status:

> *This will be addresses in token bridge (next gen, release planned end of March '23), in current ETH-TON and BSC-TON bridge won't be fixed as minor issue*

TOB-TON-53: Context sensitivity of the ; token can lead to confusion and bugs
Unresolved as of commit `e37583e5e6e8cd0aebf5142ef7d8db282f10692b`. Whitespace between two ; tokens is still parsed without a warning.

The client provided the following context for this finding's fix status:

> *We believe that visual difference between `;  ;` and `;;` are big enough to not become a serious issue. Besides with support of TF a few plugins for FunC code highlighting are developed, which will mitigate this issue further by making difference between commented and not commented code even more noticeable*

TOB-TON-54: <Retracted> Sign-confusion can lead to votes being collected incorrectly
This issue was originally reported as a finding of undetermined severity in which we speculated that sign confusion in the FunC `votes-collector` contract in the TON bridge could cause voting errors. During the fix review, we confirmed that the `udict_get?` function properly handles these sign-confusion edge cases, so we have retracted this finding.