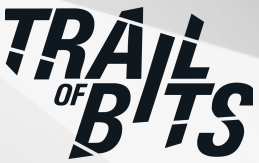

TON Foundation TVM and Fift

Final Report

July 29, 2022

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 Virtual Machine (TVM) and Fift scripting language. From July 5 to July 29, 2022, a team of 3 consultants conducted a security review of the client-provided source code, with 8 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	4
Medium	0
Low	4
Informational	4
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Data Exposure	1
Denial of Service	5
Undefined Behavior	8

Notable Findings

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.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@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	Pre-project kickoff call
July 11, 2022	Status update meeting #1
July 18, 2022	Status update meeting #2
July 25, 2022	Status update meeting #3
July 29, 2022	Delivery of report draft
July 29, 2022	Report readout meeting
TBD	Delivery of final report

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 or Fift script cause a node to expend more computational resources than the gas cost?
- Are TVM programs and Fift scripts deterministic?
- Can a maliciously crafted TVM bytecode program or Fift script be exploited to gain arbitrary code execution?
- Are the cryptographic primitives sound?

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++

Project Coverage

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

- Static analysis of the entire TON monorepo.
- Manual review of the TVM and Fift portions of the monorepo
- Fuzz testing of the bag of cells data structure and TVM
- Opcode benchmarking (CPU time versus gas cost)

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 only able to test a small fraction. We observed that many opcodes of the same family have different runtimes dependent 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.
- While we did report all findings as they were encountered (e.g., TOB-TON-4), this phase of the project was focused on the TVM and Fift.
- This phase of the project only considered the attack surface of validators through the TVM.
- The codebase would benefit from additional fuzz test harnesses, e.g., in the validator.
- Finding TOB-TON-14 has the potential to be high severity, but is currently classified with undetermined severity because there was insufficient time to confirm that it is exploitable with a proof-of-concept.

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

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.

Test Results

The results of this focused testing are detailed below.

TVM The TON virtual machine and its data structures.

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**

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	All of the high severity findings were 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.	**Strong**
Authentication / Access Controls	This phase of the project did not assess any code involving authentication or access controls.	**Not Applicable**
Complexity Management	The codebase is well organized, however, a lack of inline documentation and IDEs' inability to resolve virtual methods in all contexts sometimes hindered manual code review.	**Satisfactory**
Configuration	The TVM has many configuration options. Discovering the purpose of these options and/or the units of their values was often only possible 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).	**Satisfactory**
Data Handling	Suggestion: Include fuzz tests (see appendix)	**Satisfactory**

Decentralization	This phase of the project did not assess any code involving decentralization.	**Not Applicable**
Documentation	The high-level documentation about the TON blockchain, the TVM, and Fift are excellent. However, the codebase could benefit from more inline comments.	**Satisfactory**
Maintenance	This phase of the project did not assess the maintenance of the codebase or its deployments.	**Not Considered**
Memory Safety and Error Handling	Several findings were the result of memory safety errors.	**Weak**
Testing and Verification	Some TVM opcodes have no unit test coverage. The codebase has no automated property-based testing, fuzz testing, or formal verification.	**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	Low
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	The NOW opcode can cause consensus issues	Denial of Service	**Undetermined**

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 `std::move`'d, as is depicted in Figure 1.1. While 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 as if the `BlockDbImpl::get_out_queue_info_by_id` function that contains this bug is actually called anywhere in the code. However, if there is in fact a code path that reaches this function, then the severity of this finding would be High.

Exploit Scenario

There is a code path that 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: **Low**	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.

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.

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 having been initialized.

Exploit Scenario
The `func` utility is invoked automatically with a filename specified, for example, 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.

```
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.

On line 503, a new stack variable is returned by reference, which produces undefined behavior in C++.

The severity of this issue is undetermined because we did not exhaustively check 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 below sequences of TVM operations have been identified to 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 might 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. For a further discussion, real-world examples of the dangers of undefined behavior, and our general recommendations, see Appendix D.

Each of the following examples of TVM code that trigger undefined behavior can be triggered by running

```
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.

```
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 6.1: 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 6.2: 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 6.3: 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 6.4: 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 6.5: 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 6.6: 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 6.7: 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 6.8: Undefined behavior can be invoked on line 1354.

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 6.9: 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 6.10: Undefined behavior can be invoked on line 1626.

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 6.11: Undefined behavior can be invoked on line 1777.

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 6.12: Undefined behavior can be invoked on line 1925.

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 6.13: Undefined behavior can be invoked on line 2049.

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 and causes 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 or 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 following sequences of TVM operations have been identified to trigger undefined behavior in `crypto/common/bitstring.cpp`.

Executing code with undefined behavior in C++ implies anything can happen. Although it might seem to work as expected, results might differ depending on any factor.
When fift is build with Undefined Behavior Sanitizer, the below example can be triggered by running:

```
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.

Exploit Scenario

Blockchain nodes, running user-supplied TVM code, behave differently when the undefined behavior is triggered and causes 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 or 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 have been identified to trigger undefined behavior in `crypto/vm/tonops.cpp`.

Executing code with undefined behavior in C++ implies anything can happen. Although it might seem to work as expected, results might differ depending on any factor.
When fift is build with Undefined Behavior Sanitizer, the below example can be triggered by running:

```
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 and causes 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 or 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 have been identified to trigger undefined behavior in `crypto/vm/cells/CellBuilder.cpp`.

Executing code with undefined behavior in C++ implies anything can happen. Although it might seem to work as expected, results might differ depending on any factor.
When fift is built with the Undefined Behavior Sanitizer, the below example can be triggered by running:

```
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 and causes 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 or 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.

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 causing 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

Fift is a multipurpose language and is therefore expected to gracefully handle undefined behavior and error states. 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, make sure all the usages of `pop()` in the Fift instruction handlers are interpreted on a stack containing enough items. 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 opcode/stack state combination. 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

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.

14. The NOW opcode can cause consensus issues	
Severity: Undetermined	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-TON-14
Target: crypto/vm/tonops.cpp	

Description
The TVM includes the `NOW` opcode, which returns the current Unix time as an integer. If several validators are attempting to validate a shardchain block, they may each arrive at a different value for the `NOW` opcode.

This finding is undetermined because there was insufficient time during this phase of the assessment to test it by creating a proof-of-concept exploit.

Exploit Scenario
A smart contract is deployed that, when called, checks if `NOW` is an even or odd number. If the time is even, the contract returns a reward. If the value is odd, the contract causes an unhandled exception to be thrown. Unless the validators all validate the block on the exact same parity second, it is likely that half of the validators will sign the block and the other half will reject the block.

Recommendations
Short term, consider rounding `NOW` to a multiple of the block generation time.

Long term, consider switching the semantics of `NOW` to instead return the time the last block was added to the chain.

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

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 only implements 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&&);
```

  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
- **Only use `std::move` 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.

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    }
```

  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.

D. Risks of Undefined Behavior in C++

The C++ standard imposes no restrictions on the observable operation of a program that executes *undefined behavior*. Undefined behavior includes accessing memory outside of array bounds, null pointer dereferencing, signed integer overflow, and bit-shifting by negative values. While 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, there are certain categories of issues that they detect 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, unit and integration tests are typically the extent in which testing is performed. This type of testing allows for detecting the presence of functionality, adhering to the expected specification. However, these methods of testing do not account for other potential behaviors 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 unintentended 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 helps ease the development of testing harnesses.

By forming a system of modular components, each component is able to 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 impact 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 across all components re-using 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:

- A marketplace application allows for users to purchase listed items in buik through a JSON API.
- 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 which could lead to invalid JSON to be parsed maliciously. 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 are 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 under before 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, leading to subsequent validations failing to apply, influencing `listing.visible` and `listing.supply` and resulting in undefined behavior.

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.