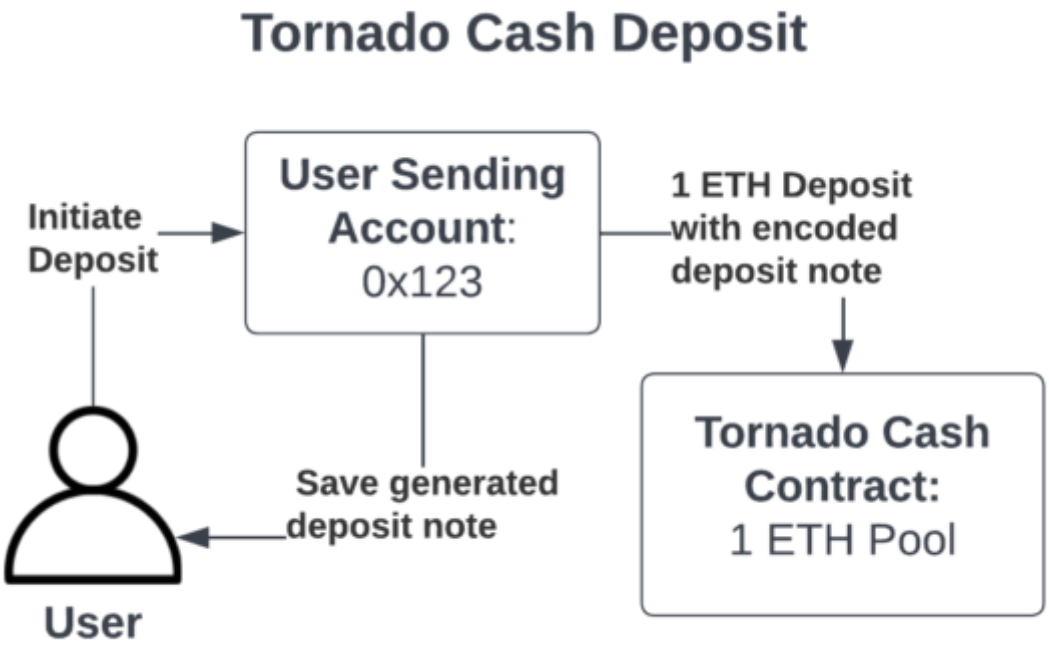
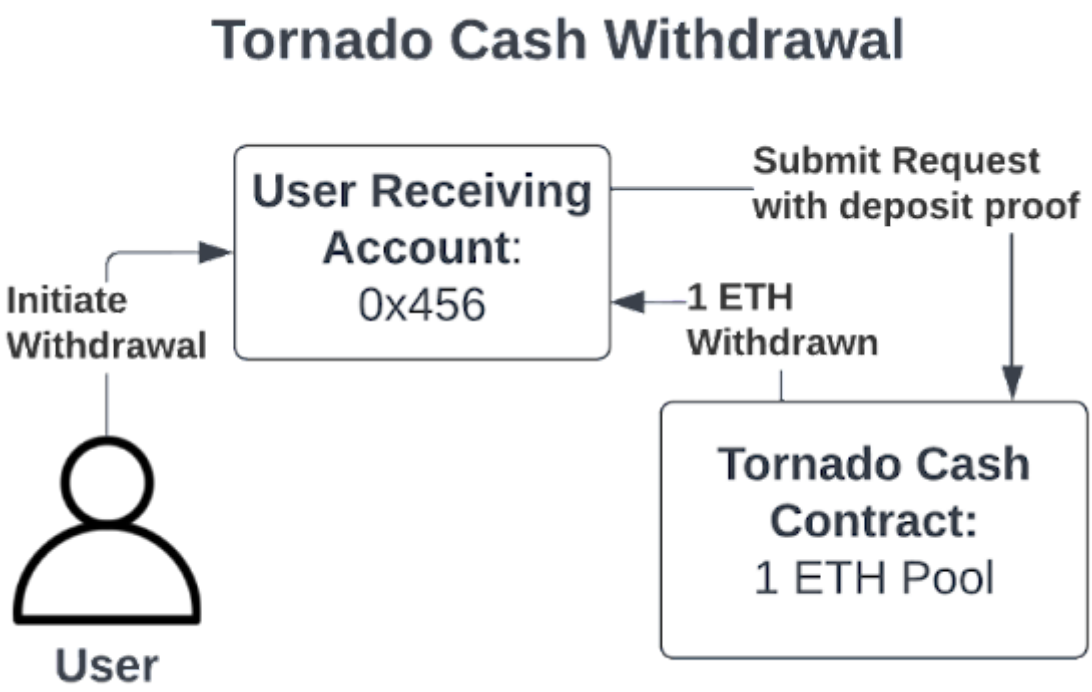
Lesson 23

Tornado Cash

Overall process

From Coincenter article





To process a deposit, Tornado.Cash generates a random area of bytes, computes it through the Pederson Hash (as it is friendlier with zk-SNARK), then send the token and the hash to the smart contract. The contract will then insert it into the Merkle tree.

To process a withdrawal, the same area of bytes is split into two separate parts: the **secret** on one side & the **nullifier** on the other side.
The nullifier is hashed.
This nullifier is a public input that is sent on-chain to get checked with the smart contract & the Merkle tree data. It avoids double spending for instance.

Thanks to zk-SNARK, it is possible to prove the hash of the initial commitment and of the nullifier without revealing any information.
Even if the nullifier is public, privacy is sustained as there is no way to link the hashed nullifier to the initial commitment. Besides, even if the information that the transaction is present in the Merkle root, the information about the exact Merkle path, thus the location of the transaction, is still kept private.

Deposits are simple on a technological point of view, but expensive in terms of gas as they need to compute the hash & update the Merkle tree. At the opposite end, the withdrawal process is complex, but cheaper as gas is only needed for the nullifier hash and the zero-knowledge proof.

TORNADO CASH Verifier Contract

Formal Verification

“Program testing can be used very effectively to show the presence of bugs but never to show their absence.” — Edsger Dijkstra

See overview resource

Introduction

Formal Verification refers to the method of mathematically proving properties of a system. To achieve this, a formal specification of the application's behaviour is created. This specification is similar to a Statement of Intended behaviour but is composed in a machine-readable language. Later, one of the available tools is used to prove or disprove the formal specification.

Correctness verification focuses on adhering to the specifications that define user interactions with smart contracts and the expected Behaviourof smart contracts when used properly.

There are two main approaches to verifying correctness:

1. Formal verification
2. Programming correctness

Formal verification techniques rely on mathematical methods, whereas programming correctness methods concentrate on ensuring that the code is accurate. In other words, programming correctness guarantees that the program does not enter an infinite loop and produces correct outputs for valid inputs.

Formal Verification vs Unit Testing

Unit testing is generally more cost-effective than other auditing methods since it is carried out during smart contract development. Comprehensive unit tests should be based on the specifications and cover the smart contract using the described use cases and functionalities. However, unit tests are as fallible as informal specifications. Even with full unit test coverage, developers might overlook certain edge cases, potentially

leading to bugs or security vulnerabilities such as overflows or unprotected functions.

Formal Verification vs Code Audits

Formal verification should not be considered a comprehensive solution or a replacement for a thorough audit. In various industries where audits are conducted, the scope of the audit includes not only the codebase but also the formal verification specifications themselves. Inaccuracies in these specifications could result in unproven application properties, allowing bugs to slip through undetected.

While it is impossible to guarantee complete safety and the absence of bugs in a smart contract, a meticulous code audit and formal verification process conducted by a reputable security firm can help identify critical, high-severity bugs that might otherwise lead to financial damage for users.

Formal Verification vs Static Analysis Tools

Static analysis tools aim to achieve the same objective as formal verification. During static analysis, a specialized program (the static analyzer) examines the smart contract or its bytecode to comprehend its Behaviourand identify unexpected scenarios such as overflows and reentrancy vulnerabilities.

However, static analysers have limitations in terms of the range of vulnerabilities they can detect. In essence, a static analyser attempts to perform formal verification using a one-size-fits-all specification that states a smart contract should not possess any known vulnerabilities. Clearly, a custom specification is more effective, as it can identify all possible vulnerabilities specific to a given smart contract.

Some tools

K Framework

The K Framework is one of the most robust and powerful language definition frameworks. It allows you to define your own programming language and provides you with a set of tools for that language, including both an executable model and a program verifier.

The KEVM provides the first machine-executable, mathematically formal, human readable and complete semantics for the EVM. The KEVM implements both the stack-based execution environment, with all of the EVM's opcodes, as well as the network's state, gas simulation, and even high-level aspects such as ABI call data.

If formal specifications of languages are defined, K Framework can handle automatic generation of various tools like interpreters and compilers. Nevertheless, this framework is very demanding (a lot of manual translations of specifications required, which are prone to error) and still suffer from some flaws (implementations of the EVM on the mainnet may not match the machine semantics for instance).

Solidity SMT Checker

See documentation
Also this useful blog

The SMTChecker module automatically tries to prove that the code satisfies the specification given by `require` and `assert` statements. That is, it considers `require` statements as assumptions and tries to prove that the conditions inside `assert` statements are always true.

The other verification targets that the SMTChecker checks at compile time are:

- Arithmetic underflow and overflow.
- Division by zero.
- Trivial conditions and unreachable code.
- Popping an empty array.
- Out of bounds index access.
- Insufficient funds for a transfer.

To enable the SMTChecker, you must select which engine should run, where the default is no engine. Selecting the engine enables the SMTChecker on all files.

The SMTChecker module implements two different reasoning engines, a Bounded Model Checker (BMC) and a system of Constrained Horn Clauses (CHC). Both engines are currently under development, and have

different characteristics. The engines are independent and every property warning states from which engine it came. Note that all the examples above with counterexamples were reported by CHC, the more powerful engine.
By default both engines are used, where CHC runs first, and every property that was not proven is passed over to BMC. You can choose a specific engine via the CLI option `--model-checker-engine {all,bmc,chc,none}` or the JSON option `settings.modelChecker.engine={all,bmc,chc,none}`

From the command line you can use

```
solc overflow.sol \
    --model-checker-targets "underflow,overflow" \
    --model-checker-engine all
```

In Remix you can add

```
pragma experimental SMTChecker;
```

to your contract, though this is deprecated.

SMT in Foundry

Example taken from Runtime verification

For the following contract

```
contract ERC20 {
    address immutable owner;
    mapping(address => uint256) public balanceOf;

    constructor() {
        owner = msg.sender;
    }

    function mint(address user, uint256 amount) external {
        require(msg.sender == owner, "Only owner can mint");
        balanceOf[user] += amount;
    }
```

```
    function transfer(address to, uint amount) external {
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
    }

    // Other functions...
}
```

We can write a test as follows

```
contract ERC20Test is Test {
    ERC20 token; // Contract under test

    address Alice = makeAddr("Alice");
    address Bob = makeAddr("Bob");
    address Eve = makeAddr("Eve");

    function setUp() public {
        token = new ERC20();
        token.mint(Alice, 10 ether);
        token.mint(Bob, 20 ether);
        token.mint(Eve, 30 ether);
    }

    function testTransfer(address from, address to, uint256
amount) public {
        vm.assume(token.balanceOf(from) >= amount);

        uint256 preBalanceFrom = token.balanceOf(from);
        uint256 preBalanceTo = token.balanceOf(to);

        vm.prank(from);
        token.transfer(to, amount);

        if(from == to) {
            assertEq(token.balanceOf(from), preBalanceFrom);
            assertEq(token.balanceOf(to), preBalanceTo);
```

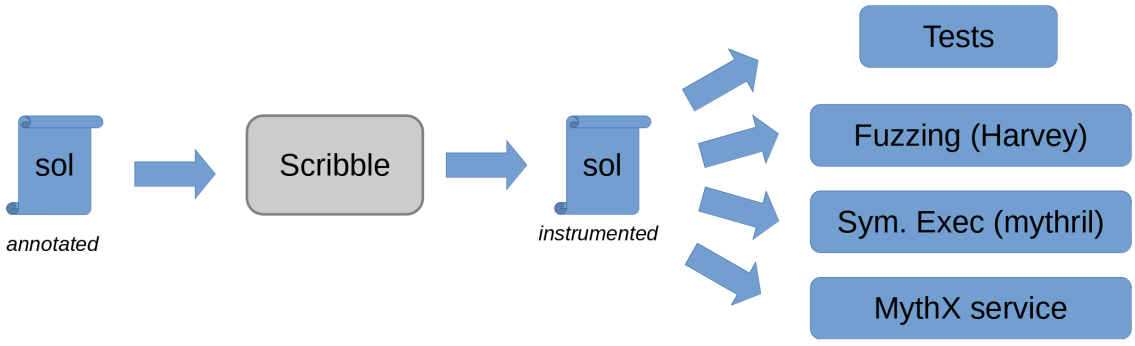
```
        } else {
            assertEq(token.balanceOf(from), preBalanceFrom -
amount);
            assertEq(token.balanceOf(to), preBalanceTo +
amount);
        }
    }
}
```

we can add the following lines to the `foundry.toml` file

```
[profile.default.model_checker]
contracts = {'../src/ERC20.sol' = ['ERC20']}
engine = 'all'
timeout = 10000
targets = ['assert']
```

Consensys Scribble

Scribble is a runtime verification tool for Solidity that transforms annotations in the Scribble specification language into concrete assertions that check the specification.
In other words, Scribble transforms existing contracts into contracts with equivalent behaviour, except that they also check properties.
With these *instrumented* contracts, you can use testing, fuzzing or symbolic execution (for example using Mythril or the MythX service) to check if your properties can be violated.



See Documentation

Installation

It is available via npm :

```
npm install -g eth-scribble
```

You add invariants to the code in the following format

```
import "Base.sol";

contract Foo is Base {

        /// #if_succeeds {:msg "P1"} y == x + 1;

        function inc(uint x) public pure returns (uint y) {

        return x+1;

        }

}
```

Scribble will then create instrumented files for you that can be used for testing.

Upcoming Changes

Lynn(Upcoming)

Height: TBD
Candidates List:

- BEP-126: Introduce Fast Finality Mechanism 24, the 2nd part.
- [BEP-?] EVM-Compatible Upgrades: Gas(EIP-3529, Warm Coinbase)

Boneh(Upcoming)

Height: TBD, Expect: Late June 2023
Candidates List:

- [BEP-?] EVM-Compatible Upgrades: New Opcode: Push0, CodeSize Limit
- BEP-126: Introduce Fast Finality Mechanism 24, the 1st part.
- BEP-174: Cross Chain Relayer Management 6
- [BEP-?] Precompile Contract For GreenField

Planck(Upcoming)

Height: 27281024, Expect: Apr-12th-2023
Candidates List:

- BEP-171: Security Enhancement for Cross-Chain Module 3
- BEP-172: Network Stability Enhancement On Slash Occur 3

BEP-126 Fast Finality

BEP-126 Proposal describes a fast finality mechanism to finalize a block, once the block has been finalized, it won't be reverted forever.

It takes several steps to finalize a block:

1. A block is proposed by a validator and propagated to other validators.
2. Validators use their BLS private key to sign for the block as a vote message.
3. Gather the votes from validators into a pool.
4. Aggregate the BLS signature if its direct parent block has gotten enough votes when proposing a new block.
5. Set the aggregated vote attestation into the extra field of the new block's header.

6. Validators and full nodes who received the new block with the direct parent block's attestation can justify the direct parent block.
7. If there are two continuous blocks have been justified, then the previous one is finalized.

The finality of a block can be achieved within two blocks in most cases, this is expected to reduce the chance of chain re-organization and stabilize the block producing further.

BEP 171

This pr will enable the upgrade of BEP171 on mainnet.
The forecasted upgrade time is 19 Apr. at 7:00 (UTC).

Cross chain bridge exploit

See analysis

Further Courses

If you would like to learn more, Encode run courses in Expert Solidity and Zero Knowledge Proofs, plus many other courses and events.

Further Resources

We have many articles / tutorials / videos available at
Extropy Foundation Site
Extropy Medium

Review

1. Introduction / Blockchain theory
2. Cryptography
3. Remix
4. Solidity language
5. Tokens and standards
6. NFTs
7. Security
8. IDEs
9. Ethers.js
10. DeFi introduction
11. Foundry
12. & 13 Gas Optimisation
13. Decentralised Storage
14. Scalability
15. AMM / DEX
16. Upgradability
17. Zero Knowledge Proofs
18. Security part 2
19. Auditing
20. DeFi continued
21. The DAO / Governance
22. Formal verification / Review