Automated formal verification: How far can we go?

Leonardo Alt & Martin Lundfall

Ethereum Foundation

leonardoalt, MrChico
{leo, martin.lundfall}@ethereum.org
leonardoalt, MartinLundfall

```
contract Token {
    mapping (address => uint) balance;
    address immutable owner;

    constructor(uint _amt) {
        owner = msg.sender;
        balance[msg.sender] = _amt;
    }

    function balanceOf(address _user) public view returns (uint) {
        return balance[_user];
    }

    function transfer(address _to, uint _amt) public {
        require(msg.sender != _to);
        require(balance[msg.sender] >= _amt);

        uint prevBal = balance[msg.sender] + balance[_to];

        balance[msg.sender] -= _amt;
        balance[_to] += _amt;

        uint postBal = balance[msg.sender] + balance[_to];

        assert(prevBal == postBal);
    }
}
```

```
contract AMM is ERC20 {
    ERC20 token0;
    ERC20 token1;

    constructor(address _token0, address _token1) {
        token0 = ERC20(_token0);
        token1 = ERC20(_token1);
    }

    // swap allows the caller to exchange amt of src for dst at a price given
    // by the constant product formula: x * y == k.
    function swap(address src, address dst, uint amt) external {
        require(src != dst, "no self swap");
        require(src == address(token0) || src == address(token1), "src not in pair");
        require(dst == address(token0) || dst == address(token1), "dst not in pair");

        uint k = token0.balanceOf(address(this)) * token1.balanceOf(address(this));

        ERC20(src).transferFrom(msg.sender, address(this), amt);

        uint out =
            ERC20(dst).balanceOf(address(this)) -
            (k / ERC20(src).balanceOf(address(this)));

        ERC20(dst).transfer(msg.sender, out);

        uint kpost = token0.balanceOf(address(this)) * token1.balanceOf(address(this));

        assert(kpost >= k);
    }
```

```
function deposit(bytes calldata pubkey,
    bytes calldata withdrawal_credentials,
    bytes calldata signature,
    bytes32 deposit_data_root
) override external payable {
    // Extended ABI length checks since dynamic types are used.
    require(pubkey.length == 48, "DepositContract: invalid pubkey length");
    require(withdrawal_credentials.length == 32, "DepositContract: invalid withdrawal_credentials length");
    require(signature.length == 96, "DepositContract: invalid signature length");
    // Check deposit amount
    require(msg.value >= 1 ether, "DepositContract: deposit value too low");
    require(msg.value % 1 gwei == 0, "DepositContract: deposit value not multiple of gwei");
    uint deposit_amount = msg.value / 1 gwei;
    require(deposit_amount <= type(uint64).max, "DepositContract: deposit value too high");

    // Emit `DepositEvent` log
    bytes memory amount = to_little_endian_64(uint64(deposit_amount));
    emit DepositEvent(
        pubkey,
        withdrawal_credentials,
        amount,
        signature,
        to_little_endian_64(uint64(deposit_count))
    );
    // Compute deposit data root (`DepositData` hash tree root)
    bytes32 pubkey_root = sha256(abi.encodePacked(pubkey, bytes16(0)));
    bytes32 signature_root = sha256(abi.encodePacked(
        sha256(abi.encodePacked(signature[:64])),
        sha256(abi.encodePacked(signature[64:], bytes32(0)))
    ));
    bytes32 node = sha256(abi.encodePacked(
        sha256(abi.encodePacked(pubkey_root, withdrawal_credentials)),
        sha256(abi.encodePacked(amount, bytes24(0), signature_root))
    ));
    // Verify computed and expected deposit data roots match
    require(node == deposit_data_root, "DepositContract: reconstructed DepositData does not match supplied deposit_data_root");
    // Avoid overflowing the Merkle tree (and prevent edge case in computing `branch`)
    require(deposit_count < MAX_DEPOSIT_COUNT, "DepositContract: merkle tree full");
    // Add deposit data root to Merkle tree (update a single `branch` node)
    deposit_count += 1;
    uint size = deposit_count;
    for (uint height = 0; height < DEPOSIT_CONTRACT_TREE_DEPTH; height++) {
        if ((size & 1) == 1) {
            branch[height] = node;
            return;
        }
        node = sha256(abi.encodePacked(branch[height], node));
        size /= 2;
    }
    // As the loop should always end prematurely with the `return` statement,
    // this code should be unreachable. We assert `false` just to be safe.
    assert(false);
}
```

- Mythril - symbolic execution
- hevm - symbolic execution, invariant fuzzing
- Echidna - invariant fuzzing
- SMTChecker - model checking
- solc-verify - model checking
- VeriSmart - model checking

Tools that claim to try to prove/break properties automatically and are publicly available.

Which tool can either prove correctness or find bugs in all the examples?

None

Automated formal verification is undecidable

Target Solidity

Cons

- A lot to encode: high level features, various data types, pointers, inheritance.
- Results rely on compiler correctness.

Pros

- Gives more structure information, for example, loops, external calls, functions.
- Can try harder problems, involving loop/contract invariants.

Model checking: SMTChecker, solc-verify, VeriSmart

Target EVM bytecode

Cons

- Not a lot of structure.
- Hard to track storage, external calls, functions.
- Needs to verify ABI encoding/decoding.

Pros

- Easier to encode.
- Results are closer to the deployed object.

Symbolic execution and fuzzing: Mythril, hevm, Echidna

Experiment:

Use each tool on each example, first automatically then tweaking parameters and writing specs taylored to the tool.

Prove functional correctness of transfer function

```
contract Token {
    mapping (address => uint) balance;
    address immutable owner;

    constructor(uint _amt) {
        owner = msg.sender;
        balance[msg.sender] = _amt;
    }

    function balanceOf(address _user) public view returns (uint) {
        return balance[_user];
    }

    function transfer(address _to, uint _amt) public {
        require(msg.sender != _to);
        require(balance[msg.sender] >= _amt);

        uint prevBal = balance[msg.sender] + balance[_to];

        balance[msg.sender] -= _amt;
        balance[_to] += _amt;

        uint postBal = balance[msg.sender] + balance[_to];

        assert(prevBal == postBal);
    }
}
```

Prove functional correctness of transfer function

- Mythril - OK
- hevm - OK
- Echidna - OK (no bugs found)
- SMTChecker - OK
- solc-verify - OK
- VeriSmart - OK

Find bug in buggy transfer function

```
contract Token {
    mapping (address => uint) balance;
    address immutable owner;

    constructor(uint _amt) {
        owner = msg.sender;
        balance[msg.sender] = _amt;
    }

    function balanceOf(address _user) public view returns (uint) {
        return balance[_user];
    }

    function transfer(address _to, uint _amt) public {
        require(msg.sender != _to);
        require(balance[msg.sender] >= _amt);

        uint prevBal = balance[msg.sender] + balance[_to];

        balance[msg.sender] += _amt;
        balance[_to] += _amt;

        uint postBal = balance[msg.sender] + balance[_to];

        assert(prevBal == postBal);
    }
}
```

Find bug in buggy transfer function

- Mythril - OK, with counterexample
- hevm - OK, with counterexample
- Echidna - OK, with counterexample
- SMTChecker - OK, with counterexample
- solc-verify - OK, no counterexample
- VeriSmart - No

AMM swap functional correctness (?)

```
contract AMM is ERC20 {
    ERC20 token0;
    ERC20 token1;

    constructor(address _token0, address _token1) {
        token0 = ERC20(_token0);
        token1 = ERC20(_token1);
    }

    // swap allows the caller to exchange amt of src for dst at a price given
    // by the constant product formula: x * y == k.
    function swap(address src, address dst, uint amt) external {
        require(src != dst, "no self swap");
        require(src == address(token0) || src == address(token1), "src not in pair");
        require(dst == address(token0) || dst == address(token1), "dst not in pair");

        uint k = token0.balanceOf(address(this)) * token1.balanceOf(address(this));

        ERC20(src).transferFrom(msg.sender, address(this), amt);

        uint out =
            ERC20(dst).balanceOf(address(this)) -
            (k / ERC20(src).balanceOf(address(this)));

        ERC20(dst).transfer(msg.sender, out);

        uint kpost = token0.balanceOf(address(this)) * token1.balanceOf(address(this));

        assert(kpost >= k);
    }
```

AMM swap functional correctness (?)

Symbolic execution and model checking tools could not prove/disprove the assertion

AMM swap functional correctness (?)

Fuzzing?

```
Fuzzing invariant
Running 1 tests for src/Amm.t.sol:TestAMM
[FAIL] invariant_k()
Replay data: '("invariant_k()", "0xf90105f870940000000000000000000000000000000
0000ff6469442997ac9251e5bb0a61f4ff790e5b991ea07fd9ba4d34e53a000000000000000000
0000000000000000000000000000000000000000000009ca000000000000000000000000000000
0000000000000000000000000000000000cf891400000000000000000000000000000000000000
99c940b7108e278c2e77e4e4f5c93d9e5e9a11ac837fcb844ac155247000000000000000000000
000000000000000000000000000000000000000000490000000000000000000000000000000000
00000000000000000000000000071a000000000000000000000000000000000000000000000000
0000000000000000c")'

Failure: invariant_k
```

```
Echidna 1.7.1
Tests found: 1
Seed: -4482172574130288672
Unique instructions: 4673
Unique codehashes: 4
Corpus size: 1
Tests
echidna_k: FAILED!

Call sequence:
1.join(0,1,1)
2.swap1(0,1)
Campaign complete, C-c or esc to exit
```

AMM swap functional correctness

Fuzzing with hevm

```
// swap allows the caller to exchange amt of src for dst at a price given
// by the constant product formula: x * y == k.
function swap(address src, address dst, uint amt) external {
    require(src != dst, "no self swap");
    require(src == address(token0) || src == address(token1), "src not in pair");
    require(dst == address(token0) || dst == address(token1), "dst not in pair");

    KPrev = token0.balanceOf(address(this)) * token1.balanceOf(address(this));

    ERC20(src).transferFrom(msg.sender, address(this), amt);

    uint out =
        ERC20(dst).balanceOf(address(this)) -
        (KPrev / ERC20(src).balanceOf(address(this)));

    ERC20(dst).transfer(msg.sender, out);

    KPost = token0.balanceOf(address(this)) * token1.balanceOf(address(this));
}

function invariant_k() public view {
    assert(KPost >= KPrev);
}

function kprev() public view returns (uint) {
    return KPrev;
}

function kpost() public view returns (uint) {
    return KPost;
}
```

```
contract TestAMM is DSTest {
    MintableERC20 token0;
    MintableERC20 token1;
    AMM pair;

    User user0;
    User user1;
    User user2;

    function targetContracts() public view returns (address[] memory) {
        address[] memory addrs = new address[](3);
        addrs[0] = address(user0);
        addrs[1] = address(user1);
        addrs[2] = address(user2);
        return addrs;
    }

    function setUp() public {
        token0 = new MintableERC20();
        token1 = new MintableERC20();
        pair = new AMM(address(token0), address(token1));

        user0 = new User(token0, token1, pair);
        user1 = new User(token0, token1, pair);
        user2 = new User(token0, token1, pair);

        token0.approve(address(pair), type(uint).max);
        token1.approve(address(pair), type(uint).max);

        uint amt = type(uint).max / 24;

        token0.mint(address(user0), amt);
        token0.mint(address(user1), amt);
        token0.mint(address(user2), amt);

        token1.mint(address(user0), amt);
        token1.mint(address(user1), amt);
        token1.mint(address(user2), amt);
    }

   function invariant_k() public view {
       pair.invariant_k();
   }
}
```

AMM swap functional correctness

Fuzzing with Echidna

```
contract TestAMM {
    MintableERC20 token0;
    MintableERC20 token1;
    AMM pair;

    User[3] users;

    constructor() {
        token0 = new MintableERC20();
        token1 = new MintableERC20();
        pair = new AMM(address(token0), address(token1));

        users[0] = new User(token0, token1, pair);
        users[1] = new User(token0, token1, pair);
        users[2] = new User(token0, token1, pair);

        token0.approve(address(pair), type(uint).max);
        token1.approve(address(pair), type(uint).max);

        uint amt = type(uint).max / 24;

        token0.mint(address(users[0]), amt);
        token0.mint(address(users[1]), amt);
        token0.mint(address(users[2]), amt);

        token1.mint(address(users[0]), amt);
        token1.mint(address(users[1]), amt);
        token1.mint(address(users[2]), amt);
    }

    function echidna_k() public returns (bool) {
        return pair.kpost() >= pair.kprev();
    }
```

AMM swap functional correctness

```
function swap(address src, address dst, uint amt) external {
    require(src != dst, "no self swap");
    require(src == address(token0) || src == address(token1), "src not in pair");
    require(dst == address(token0) || dst == address(token1), "dst not in pair");

    KPrev = token0.balanceOf(address(this)) * token1.balanceOf(address(this));

    ERC20(src).transferFrom(msg.sender, address(this), amt);

    uint out =
        ERC20(dst).balanceOf(address(this)) -
        (KPrev / ERC20(src).balanceOf(address(this)) + 1);

    ERC20(dst).transfer(msg.sender, out);

    KPost = token0.balanceOf(address(this)) * token1.balanceOf(address(this));
}
```

Model checkers still cannot prove correctness, but fuzzers could not find any other problems.

```
Fuzzing invariant
Running 1 tests for src/Amm.t.sol:TestAMM
[PASS] invariant_k() (runs: 100, depth: 20)
```

```
Echidna 1.7.1
Tests found: 1
Seed: 880666784831901063
Unique instructions: 4724
Unique codehashes: 4
Corpus size: 2
Tests
echidna_k: fuzzing (1280/50000)
```

Prove function correctness of deposit

```
function deposit(bytes calldata pubkey,
    bytes calldata withdrawal_credentials,
    bytes calldata signature,
    bytes32 deposit_data_root
) override external payable {
    // Extended ABI length checks since dynamic types are used.
    require(pubkey.length == 48, "DepositContract: invalid pubkey length");
    require(withdrawal_credentials.length == 32, "DepositContract: invalid withdrawal_credentials length");
    require(signature.length == 96, "DepositContract: invalid signature length");
    // Check deposit amount
    require(msg.value >= 1 ether, "DepositContract: deposit value too low");
    require(msg.value % 1 gwei == 0, "DepositContract: deposit value not multiple of gwei");
    uint deposit_amount = msg.value / 1 gwei;
    require(deposit_amount <= type(uint64).max, "DepositContract: deposit value too high");

    // Emit `DepositEvent` log
    bytes memory amount = to_little_endian_64(uint64(deposit_amount));
    emit DepositEvent(
        pubkey,
        withdrawal_credentials,
        amount,
        signature,
        to_little_endian_64(uint64(deposit_count))
    );
    // Compute deposit data root (`DepositData` hash tree root)
    bytes32 pubkey_root = sha256(abi.encodePacked(pubkey, bytes16(0)));
    bytes32 signature_root = sha256(abi.encodePacked(
        sha256(abi.encodePacked(signature[:64])),
        sha256(abi.encodePacked(signature[64:], bytes32(0)))
    ));
    bytes32 node = sha256(abi.encodePacked(
        sha256(abi.encodePacked(pubkey_root, withdrawal_credentials)),
        sha256(abi.encodePacked(amount, bytes24(0), signature_root))
    ));
    // Verify computed and expected deposit data roots match
    require(node == deposit_data_root, "DepositContract: reconstructed DepositData does not match supplied deposit_data_root");
    // Avoid overflowing the Merkle tree (and prevent edge case in computing `branch`)
    require(deposit_count < MAX_DEPOSIT_COUNT, "DepositContract: merkle tree full");
    // Add deposit data root to Merkle tree (update a single `branch` node)
    deposit_count += 1;
    uint size = deposit_count;
    for (uint height = 0; height < DEPOSIT_CONTRACT_TREE_DEPTH; height++) {
        if ((size & 1) == 1) {
            branch[height] = node;
            return;
        }
        node = sha256(abi.encodePacked(branch[height], node));
        size /= 2;
    }
    // As the loop should always end prematurely with the `return` statement,
    // this code should be unreachable. We assert `false` just to be safe.
    assert(false);
```

Prove function correctness of deposit

No tool could prove that the assertion is not reachable automatically.

Modified version for hevm

```
function deposit(
    bytes16 pubkey_0, bytes16 pubkey_1, bytes16 pubkey_2,
    bytes32 withdrawal_credentials,
    bytes32 signature_0, bytes32 signature_1, bytes32 signature_2,
    bytes32 deposit_data_root
) override external payable {
    // Check deposit amount
    require(msg.value >= 1 ether, "DepositContract: deposit value too low");
    require(msg.value % 1 gwei == 0, "DepositContract: deposit value not multiple of gwei");
    uint deposit_amount = msg.value / 1 gwei;
    require(deposit_amount <= type(uint64).max, "DepositContract: deposit value too high");

    // Emit `DepositEvent` log
    bytes memory amount = to_little_endian_64(uint64(deposit_amount));
    emit DepositEvent(
        pubkey_0, pubkey_1, pubkey_2,
        withdrawal_credentials,
        amount,
        signature_0, signature_1, signature_2,
        to_little_endian_64(uint64(deposit_count))
    );

    // Compute deposit data root (`DepositData` hash tree root)
    bytes32 pubkey_root = sha256(abi.encodePacked(pubkey_0, pubkey_1, pubkey_2, bytes16(0)));
    bytes32 signature_root = sha256(abi.encodePacked(
        signature_0, signature_1, signature_2, bytes32(0)
    ));
    bytes32 node = sha256(abi.encodePacked(
        sha256(abi.encodePacked(pubkey_root, withdrawal_credentials)),
        sha256(abi.encodePacked(amount, bytes24(0), signature_root))
    ));

    // Verify computed and expected deposit data roots match
    require(node == deposit_data_root, "DepositContract: reconstructed DepositData does not match supplied deposit_data_root");
    // Avoid overflowing the Merkle tree (and prevent edge case in computing `branch`)
    require(deposit_count < MAX_DEPOSIT_COUNT, "DepositContract: merkle tree full");

    // Add deposit data root to Merkle tree (update a single `branch` node)
    deposit_count += 1;
    uint size = deposit_count;
    for (uint height = 0; height < DEPOSIT_CONTRACT_TREE_DEPTH; height++) {
        if ((size & 1) == 1) {
            branch[height] = node;
            return;
        }
        node = sha256(abi.encodePacked(branch[height], node));
        size /= 2;
    }
    // As the loop should always end prematurely with the `return` statement,
    // this code should be unreachable. We assert `false` just to be safe.
    assert(false);
}
```

hevm results

```
Running Deposit contract
checking postcondition...
Q.E.D.
Explored: 295 branches without assertion violations
```

Modified version for SMTChecker

```
// Add deposit data root to Merkle tree (update a single `branch` node)
deposit_count += 1;
uint size = deposit_count;
for (uint height = 0; height < DEPOSIT_CONTRACT_TREE_DEPTH; height++) {
    //if ((size & 1) == 1) {
    if ((size % 2) == 1) {
        branch[height] = node;
        return;
    }
    node = sha256(abi.encodePacked(branch[height], node));
    size /= 2;
}
// As the loop should always end prematurely with the `return` statement,
// this code should be unreachable. We assert `false` just to be safe.
assert(false);
```

+ use non default Horn solver Eldarica via solc-js' SMT callback

+ use Eldarica's abstract:off option

SMTChecker's inductive invariant for the loop before the assertion

```
(define-fun block_54_for_header_deposit_367_406_0 ((A Int) (B Int) (C abi_type) (D crypto_type) (E tx_type) (F state_type) (G Int) (H Int) (I fixedbytes_array_tuple) (J Int) (K fixedbytes_array_tuple) (L bytes_tuple) (M bytes_tuple) (N bytes_tuple) (O Int) (P state_type) (Q Int) (R Int) (S fixedbytes_array_tuple) (T Int) (U fixedbytes_array_tuple) (V bytes_tuple) (W bytes_tuple) (X bytes_tuple) (Y Int) (Z Int) (A1 bytes_tuple) (B1 Int) (C1 Int) (D1 Int) (E1 Int) (F1 Int)) Bool (and (and (and (and (and (and (and (and (= P F) (= Q G)) (= R H)) (= S I)) (= U K)) (= V L)) (= W M)) (= X N)) (and (and (and (and (and (= A 0) (= O 0)) (= Y 0)) (= B1 0)) (= C1 0)) (and (and (and (= (bytes_tuple_accessor_length L) 48) (= (bytes_tuple_accessor_length M) 32)) (= (bytes_tuple_accessor_length N) 96)) (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (or (and (and (= (+ J (* (- 2147483648) E1)) (- 1)) (= F1 31)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0))) (and (and (= (+ J (* (- 1073741824) E1)) (- 1)) (= F1 30)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 536870912) E1)) (- 1)) (= F1 29)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 268435456) E1)) (- 1)) (= F1 28)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 134217728) E1)) (- 1)) (= F1 27)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 67108864) E1)) (- 1)) (= F1 26)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 33554432) E1)) (- 1)) (= F1 25)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 16777216) E1)) (- 1)) (= F1 24)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 8388608) E1)) (- 1)) (= F1 23)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 4194304) E1)) (- 1)) (= F1 22)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 2097152) E1)) (- 1)) (= F1 21)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 1048576) E1)) (- 1)) (= F1 20)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 524288) E1)) (- 1)) (= F1 19)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 262144) E1)) (- 1)) (= F1 18)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 131072) E1)) (- 1)) (= F1 17)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 65536) E1)) (- 1)) (= F1 16)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 32768) E1)) (- 1)) (= F1 15)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 16384) E1)) (- 1)) (= F1 14)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 8192) E1)) (- 1)) (= F1 13)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 4096) E1)) (- 1)) (= F1 12)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 2048) E1)) (- 1)) (= F1 11)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 1024) E1)) (- 1)) (= F1 10)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 512) E1)) (- 1)) (= F1 9)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 256) E1)) (- 1)) (= F1 8)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 128) E1)) (- 1)) (= F1 7)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 64) E1)) (- 1)) (= F1 6)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 32) E1)) (- 1)) (= F1 5)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 16) E1)) (- 1)) (= F1 4)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 8) E1)) (- 1)) (= F1 3)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 4) E1)) (- 1)) (= F1 2)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (<= E1 4294967295)) (>= E1 0)))) (and (and (= (+ J (* (- 2) E1)) (- 1)) (= F1 1)) (and (and (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (>= (- E1 J) (- 4294967294))) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0)) (>= E1 0)))) (and (and (= (- J E1) (- 1)) (= F1 0)) (and (and (and (and (and (and (and (<= (msg.value E) 18446744073709551615999999999) (>= (msg.value E) 1000000000000000000)) (<= J 4294967294)) (>= J 0)) (<= T 4294967295)) (>= T 1)) (<= D1 115792089237316195423570985008687907853269984665640564039457584007913129639935)) (>= D1 0))))))))
```

Thank you!