File 1 of 1

HITB SECCONF Amsterd4m and ConsenSys Dilig3nce bring you

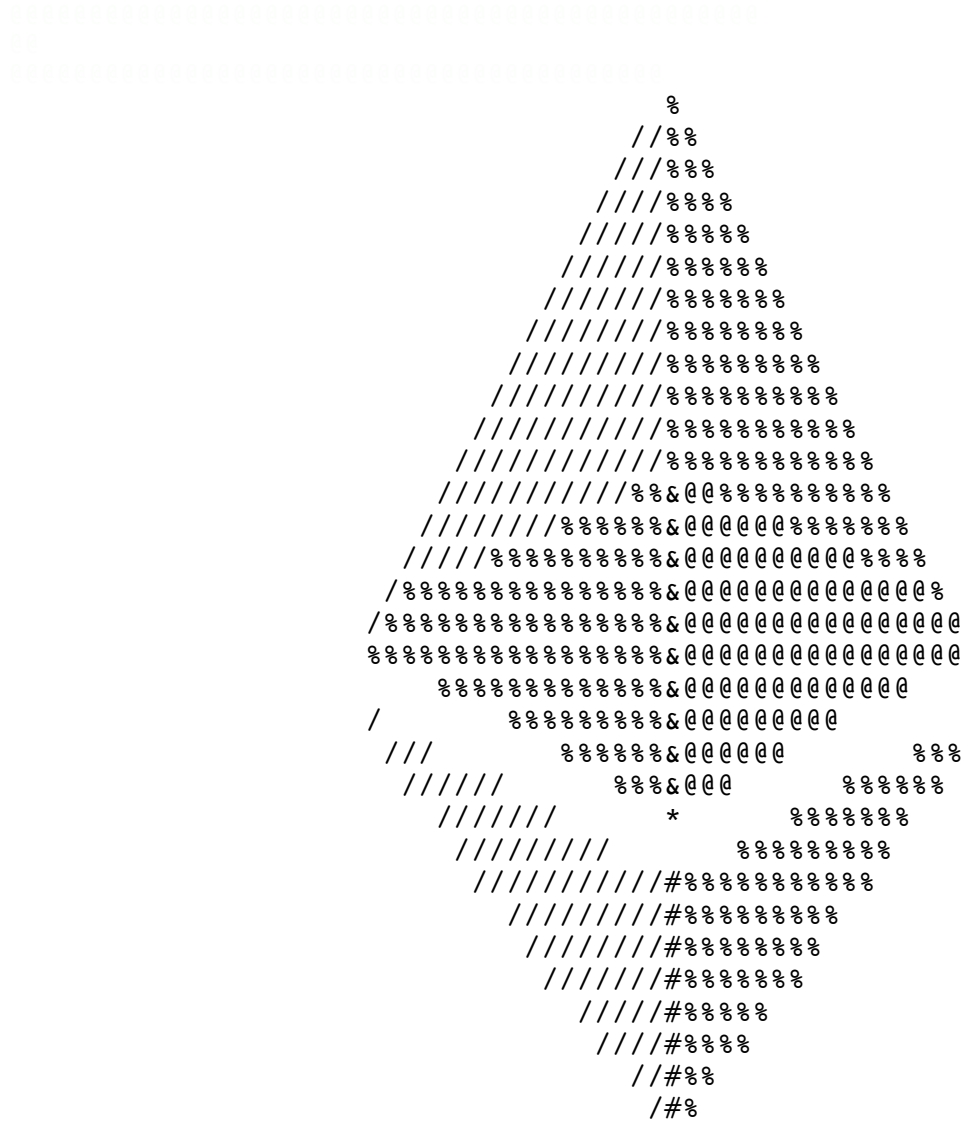
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Smashing Ethereum Smart Contracts for Fun and Real Profit

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Bernhard Mueller

bernhard.mueller@consensys.net



Introduction

During my first year in university, I discovered Phrack magazine and the 1,746 infamous lines of ASCII text titled "Smashing the Stack for Fun and Profit" [1]. Up until that point, I'd been on a trajectory to becoming a web designer, but Aleph One's legendary introduction to buffer overflow exploits inspired me (like countless others) to specialize in computer security instead. What followed was an exciting time of learning and discovery. For the next few years, nothing was more magical than seeing that shellcode executed, seeing database contents revealed bit-by-bit through a blind SQL injection attack, and crashing a whole Wi-Fi network with a single spoofed RIP packet.

Exploring distributed ledgers and the Ethereum "world computer" reminded me of those early days. The Ethereum blockchain supports smart contracts, quasi-Turing-complete programs that run in a stack-based virtual machine. Because we haven't learned much since 1996, most of these contracts are developed in a programming language that allows the introduction of a variety of bugs.

This time around there's one crucial difference though. In the early days, bug bounty programs didn't exist, and zero-day vulnerabilities were dumped on mailing lists just for the so-called lulz, so unless you had rather dubious connections, the only profit to be made was gaining the respect of other security researchers. Hack a smart contract however, and you might see some *actual* money.

This paper introduces Mythril [2], a security analysis tool for Ethereum smart contracts, and its symbolic execution backend LASER-Ethereum [3]. The first part of the paper explains applications of symbolic execution and constraint solving in smart contract security analysis and verification. The second part showcases the use of symbolic analysis, static analysis, and control flow checking to discover real-world issues.

The work is not groundbreaking by any measure, but hopefully it'll help to make the Ethereum ecosystem a little bit safer. At the very least, I hope reading the paper is as much fun as writing it was. If it inspires one or two readers to learn more about smart contract security, even better! With that in mind, I've tried to make the paper as accessible as possible. Most examples can be reproduced using Mythril and the supplemental materials available on Github [4]. If you have questions, feel free to post an issue or ask on our Gitter channel [5].

I'd like to thank Mario Alvarez, Heaven Hodges, Tom Lindeman, John Mardlin, Gonçalo Sá and Gerhard Wagner for corrections and feedback, and the ConsenSys Diligence team for their unrelenting support.

Table of Contents
~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~

Symbolic Analysis of Ethereum Bytecode

Although our industry hasn't had much success in eradicating software bugs, we have done an excellent job at inventing a colorful portfolio of techniques for *detecting* them. A few that come to mind are SAST, DAST, IAST and RASP (if you don't know what those acronyms mean, ask a CISSP or CEH). Another approach that has become popular in the 2000s is symbolic execution. With this approach, program inputs are assumed to be symbols that represent arbitrary input values. During the symbolic execution run, the interpreter keeps track of the program states it encounters and collects constraints on inputs from predicates encountered in branch instructions. Every execution path discovered can be expressed as a propositional formula. The resulting representation of the program states and control flow can be used to prove certain properties of the program, determine reachability of error states, and perform various types of security analysis.

While symbolic execution is very powerful in theory, it has some drawbacks in real work applications. For one, discovering all feasible execution paths in a reasonably complex program is very memory-intensive and time-consuming. Operating system features (such as file systems, sockets and multi-threading) are also difficult to model. The Ethereum Virtual Machine, however, is simple compared to a desktop or mobile operating system, and this simplicity makes it possible to achieve 100% path coverage with typical smart contracts.

Modeling Transaction Execution

Ethereum is best described as a transaction-based distributed state machine: At any point in time, Ethereum nodes agree on a shared world state that is modified over time by transactions accepted into the blockchain. The world state at any given time is the result of all state modifications since the genesis state. Ethereum accounts may contain code ("smart contracts") that is executed in the Ethereum Virtual Machine (EVM) whenever a transaction is received.

The Ethereum yellow paper [6] formally specifies the Ethereum state machine and virtual machine. Written in ancient Greek symbols, the paper is notoriously difficult to understand, but the machines it describes are actually fairly simple. This simplicity becomes obvious once one looks at an actual implementation (for an easier-to-understand description, I recommend reading Micah Dameron's beige paper [7]).

LASER-Ethereum [3] is a symbolic interpreter for Ethereum bytecode. Given one or more smart contract accounts as input, it returns a set of abstract program states. A *state* consists of the set of values that the virtual machine variables (such as the program counter, virtual machine stack, and

account balances) take at a particular point during execution.

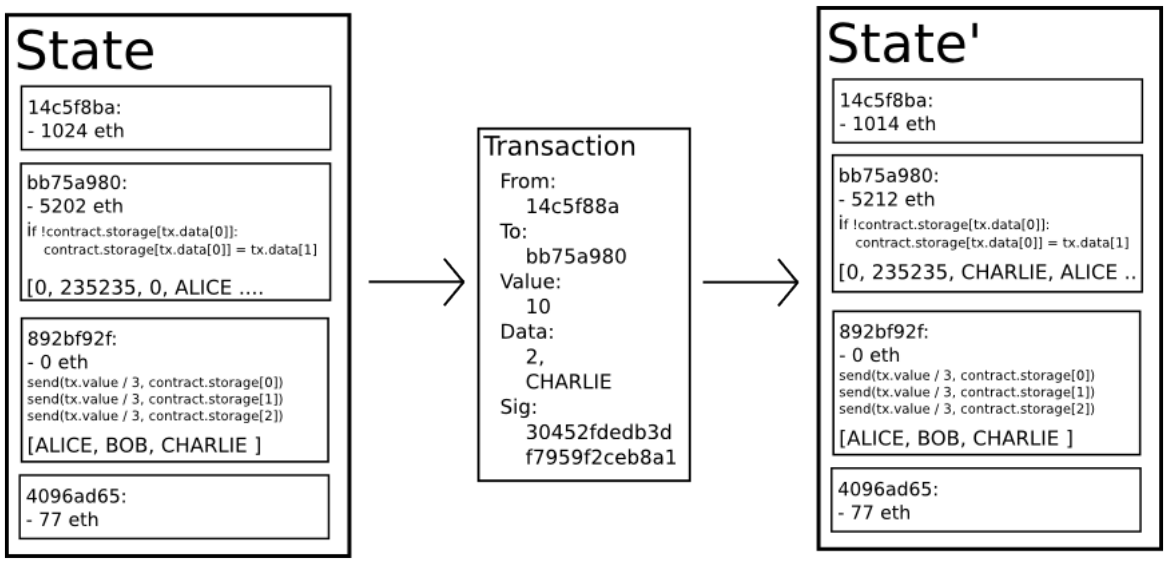


Figure 1: State transition diagram from the Ethereum White Paper [8]

The yellow paper describes three sets of state variables:

- World State (σ): A mapping of Ethereum addresses to accounts, which include account storage and balances.
- Machine State (μ): The program counter, memory, and stack of the virtual machine.
- Execution Environment (I): Variables relevant to the transaction that is currently executing (caller address, transaction value, and so on).

In LASER, the overall state is represented by the GlobalState Python object that holds the machine state, environment, and world state (Figure 2).

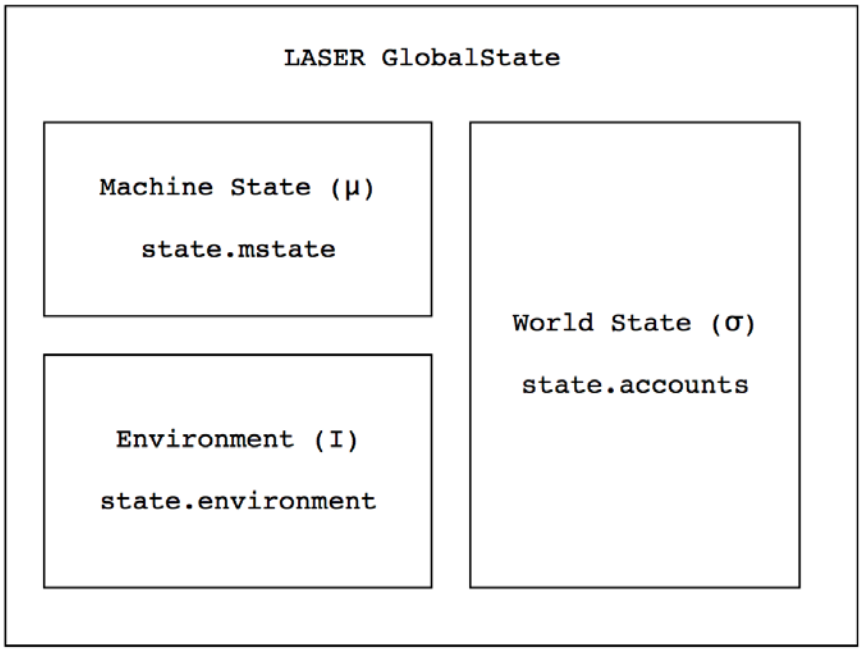


Figure 2: The LASER GlobalState object

Obtaining the space of program states with LASER is as simple as initializing a contract account and calling the sym_exec() method:

```
--------------------------------------------------------------------------
>>> account = svm.Account(address, disassembly, "MyContract")
>>> accounts = {address: account}
>>>
>>> laser = svm.LaserEVM(accounts)
>>> statespace = laser.sym_exec(address)
>>>
>>> statespace.nodes[0].states
[<laser.ethereum.svm.GlobalState object at 0x10ae1e978>,
<laser.ethereum.svm.GlobalState object at 0x10ae1ea58>,...]
--------------------------------------------------------------------------
```

The LASER GlobalState object contains three members: world state, machine state, and environment (denoted in the yellow paper by σ, μ, and I, respectively). Let's look at the components of the state in more detail.

World State (σ)

According to the yellow paper [6], the world state comprises of a mapping from Ethereum addresses to accounts, each of which has the following four fields:

- nonce: A scalar value equal to the number of transactions sent from the account or, given accounts with associated code, the number of contract creations made by the account. The nonce is denoted by $\sigma[a]_n$.
- balance: A scalar value equal to the number of Wei owned by the mapped address. Denoted by $\sigma[a]_b$.
- storageRoot: A 256-bit hash of the root node of a Merkle Patricia tree that encodes the account's storage contents (a mapping between 256-bit integer values), denoted by $\sigma[a]_s$.
- codeHash: The hash of the mapped account's EVM code. This is the code that gets executed if the mapped address receives a message call; it is immutable and thus, unlike all other fields, cannot be changed after construction. All such code fragments are contained in the state database under their corresponding hashes for future retrieval. This hash is formally denoted by $\sigma[a]_c$, and the code may thus be denoted by b if KEC(b) = $\sigma[a]_c$.

In LASER, the world state is represented by a dictionary that maps hex-encoded addresses to account objects:

```
--------------------------------------------------------------------------
>>> state.accounts['0x0000000000000000000000000000000000000000'].as_dict()
{'nonce': 0,
 'code': <mythril.disassembler.disassembly.Disassembly object at
0x106413940>,
  'balance': balance,
  'storage': {}
}
--------------------------------------------------------------------------
```

The implementation is faithful to the yellow paper except for the codeHash field. Instead of a hash, the account object contains the code itself in the form of a Mythril Disassembly object. Such an object can be generated from bytecode or source code with a few lines of Python code.

```
--------------------------------------------------------------------------
from mythril.ether.soliditycontract import SolidityContract

contract = SolidityContract("solidity_examples/underflow.sol", "Under")
disassembly = contract.get_disassembly()
--------------------------------------------------------------------------
```

Machine State (μ)

Quoting the yellow paper [6], "the machine state μ is defined as the tuple (g, pc, m, i, s), whose elements are the gas available, the program counter pc ∈ P256, the memory contents, the active number of words in memory (counting continuously from position 0), and the stack contents."

In LASER, the machine state is represented by GlobalState.mstate:

```
>>> state.mstate.as_dict()
{'pc': 0,
 'stack': [],
 'memory': [],
 'memsize': 0,
 'gas': 10000000
}
```

Execution Environment (I)

The execution environment consists of the following variables [6]:

- I_a, the address of the account that owns the executing code.

- I_o, the sender address of the transaction that originated the execution.

- I_p, the price of gas in the transaction that originated the execution.

- I_d, the input byte array for the execution. If the execution agent is a transaction, Id is the transaction data.

- I_s, the address of the account that caused the code execution. If the execution agent is a transaction, the transaction sender is.

- I_v, the value in Wei passed to this account as part of the procedure that execution belongs to. If the execution agent is a transaction, Iv is the transaction value.

- I_b, the byte array represented by the machine code to be executed.

- I_H, the block header of the present block.

- I_e, the depth of the present message call or contract-creation (i.e., the number of CALLs or CREATEs being executed).

In LASER, the environment is represented by GlobalState.environment:

```
>>> state.environment.as_dict()
  {'active_account': <laser.ethereum.svm.Account object at 0x1064a4780>,
   'sender': caller,
   'calldata': [],
```

```
    'gasprice': gasprice,
    'callvalue': callvalue,
    'origin': origin,
    'calldata_type': <CalldataType.SYMBOLIC: 2>}
--------------------------------------------------------------------------------
```

LASER doesn't yet have extensive documentation, but its README.md has a basic how-to (you can also read the source code which clocks in at less than 1,500 LoC).

Control Flow

LASER organizes program states via a control flow graph (Figure 3). Each node of the graph represents a basic block of code that is executing. Every node has a set of path conditions. A list of edges between nodes and the constraint on each edge is also provided. The control flow graph (CFG) isn't strictly needed for symbolic analysis, but it enables additional useful types of analysis and the rendering of awesome visuals.

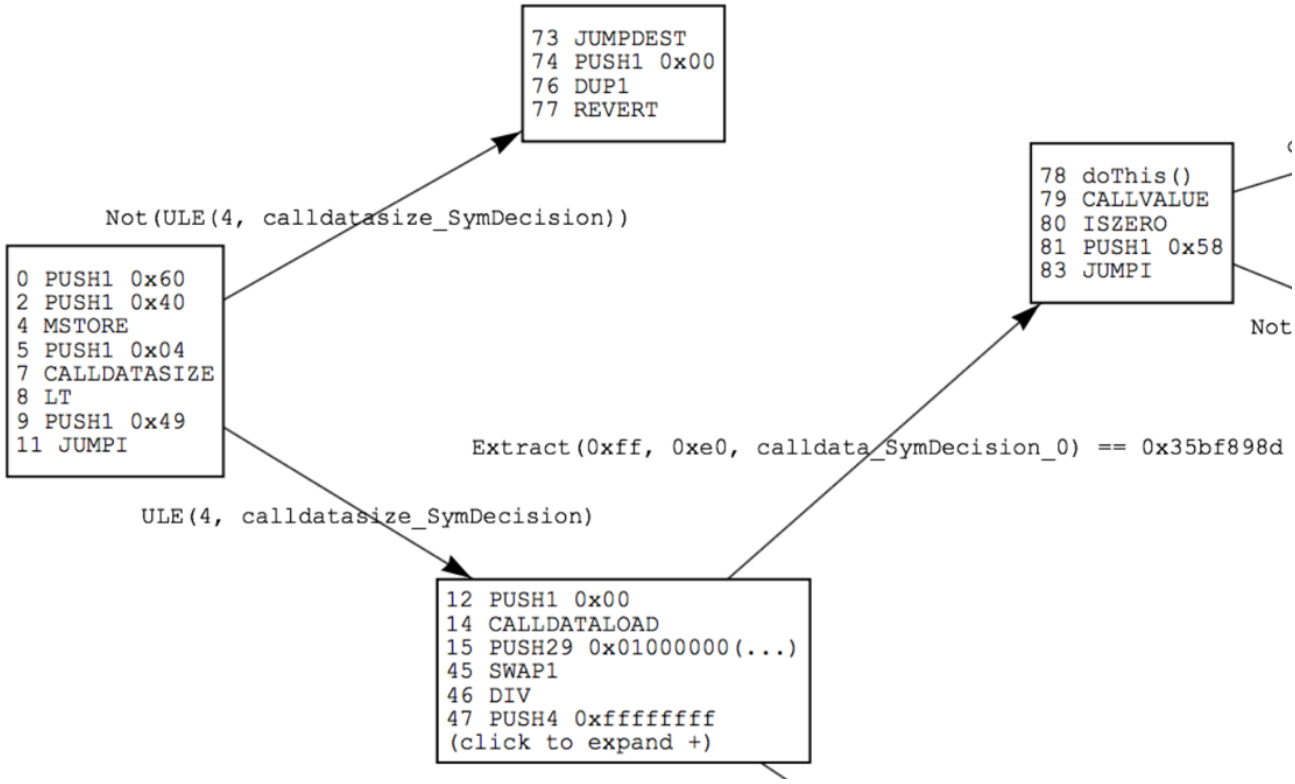


Figure 3: Program states mapped along a control flow graph. Nodes represent basic blocks of code; Each line in a node represents a program state. Edges are annotated with path constraints.

A key concept in symbolic analysis is the *path formula*, a logical formula composed of the constraints on a given path. The path formula of a given node is the result of the logical AND of all edges on the path to that node. The path formula for reaching the function doThis() in Figure 3 is

```
calldatasize < 4 ^ calldata[0:4] = 0x35bf8089d
```

Note that 0x35bf8089d corresponds to the function signature hash for doThis().

********** PRO TIP **********

You can quickly calculate a function hash by using Mythril's hash utility:

```
$ myth --hash "doThis()"
0x35bf898d
```

****

SMT Solving and Formal Proofs

Using LASER, we can represent smart contract execution as a space of states and path formulas in propositional logic. Obviously, this in itself is a mind-blowing achievement, but how is it useful for security analysis?

We can produce statements ("theorems") using propositional logic and attempt to prove or disprove those statements within the space of abstract states discovered. We could, for example, ask whether there is feasible path to a particular program. Questions like this can be expressed as Boolean satisfiability (SAT) problems, for example: Is it possible to satisfy the formula a ∧ (b ∨ c) ∧ ¬ c ∧ ¬ (b ∧ d)?

To solve these problems, we use automated reasoning tools called SMT Solvers. A solver can check the satisfiability of logical formulas over one or more *theories*. The EVM computes using 256-bit bit-vectors, so we can use the bit-vector theory to reason about those computations.

Fortunately, being engineers, we don't need to worry about *how* the solver actually computes solutions. LASER uses Z3, a popular theorem prover from Microsoft Research. In the following sections, we'll throw examples of increasing complexity at LASER and the Z3 Solver.

Asserting the Obvious

The easiest way to understand the solving process is by example. Let's first look at a simple functional correctness check using assertions.

Solidity provides an assert() function that can be used to assert invariants. Invariants are conditions that are expected to always hold during run time, regardless of the environment, input variables, and initial state.

The following smart contract contains two assertions.

```
contract Assertions {

    function assertion1(uint256 input) {
        assert(input * 4 < 1024);
    }

    function assertion2(uint256 input) {
        if (input > 256) {
            throw;
        }

        assert(input * 4 <= 1024);
    }

}
```

At the end of each function, (input * 4 < 1024) is asserted. The compiler translates the assert function call into a conditional jump that leads into an invalid opcode (0xfe) if the condition isn't met. Therefore, if the condition (input * 4 < 1024) is violated during a program run, execution will terminate with an "invalid opcode" exception and the transaction will be rolled back.

Looking at the source, it seems obvious that the assertion in the function assertion2 always holds. But how do we *prove* this? One approach is to run the code with all possible input values and environment configurations and observe whether the exception is triggered. But there's a catch: The number of possible values that a single 256-bit integer can take is approximately equal to the number of hydrogen atoms in the observable universe. That's quite a long fuzzing run. A better approach is to logically prove that the instruction can never be executed.

The exception is located at PC address 171 (Figure 4). Suppose M is the set of all machine states in which the program counter value is 171 (note that there could be zero or more states in which this is the case, but here we have exactly one):

$$M = \{\mu: \mu_{pc} = 171\}$$

We would like to prove that for all states $\mu \in M$ there exists no pair of initial state and environmental variables (σ, I) that satisfies the path formula $P_\mu(\sigma, I)$ (which represents the conditions under which μ can be reached).

$$\forall \mu \in M \rightarrow \neg\exists(\sigma,I)\ P_\mu(\sigma, I)$$

Figure 4 shows the control flow graph for the function assertion2. Because the graph is too large to fit into this paper, only the portion representing the if-statement is shown - the full version is available in the supplemental materials [4].

As previously discussed, the path formula of a given node is the result of the logical AND of all edges along the path to that node. For the node containing the exception (ASSERT_FAIL at address 171), the AND operation yields the following path formula:

$$I_d[0:4] = 0xe166a663 \ \hat{}\ I_v = 0 \ \hat{}\ UINT256(I_d[4:36]) < 0x100 \ \hat{}\ 4 * UINT256(I_d[4:36]) > 0x400$$

where I_d is a byte array containing the input data and I_v is the call value.

The leftmost two conditions in this formula represent the function signature and call value. We can assume that they are always satisfiable. The right half of the formula can be simplified to:

$$(input < 0x400) \ \hat{}\ \neg(input < 0x400)$$

As the wise Aristotle noted, a proposition *Q* and its negation $\neg Q$ (not-*Q*) cannot both be true, and advances in quantum mechanics notwithstanding, this is still the consensus. By showing a contradiction in the path formula, we have shown that the exception in assertion2() can never be executed.

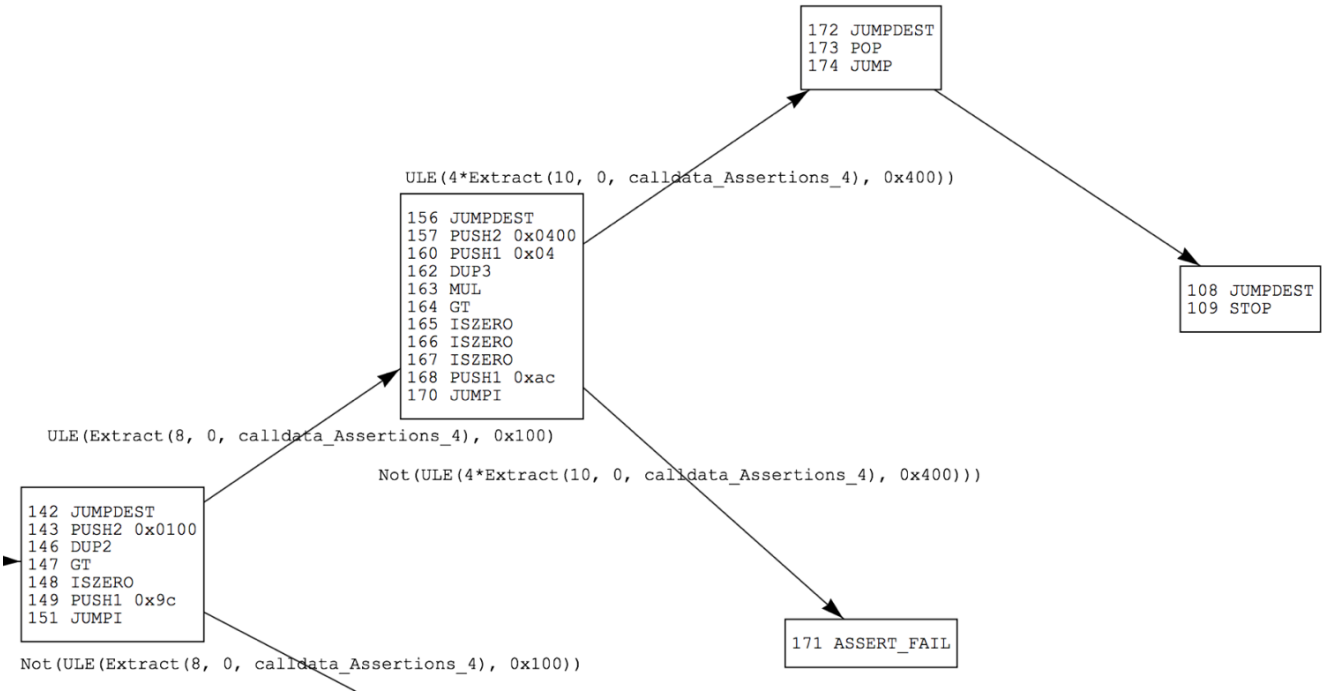


Figure 4: CFG of the function assertion2(uint256)

********** PRO TIP **********

The CFGs in this paper were produced via the Mythril command line tool. You can create your own CFGs by running

```
$ myth -g graph.html assertions.sol
```

If you prefer that your graphs look like the ones in this paper, add the following secret command line flag:

```
$ myth --phrack -g graph.html assertions.sol
```

****

What about the assertion in the function assertion1? As in the previous case, only one path to the exception state at μ_{pc} = 189 exists. The logical conjunction of all the constraints along the edges gives the following path formula:

$$(I_d[0:4] = 0xe166a663) \wedge (I_v = 0) \wedge (UINT256(I_d[4:36]) > 0x400)$$

This time, it is possible to satisfy the formula by, for example, choosing the following values (the dot "." indicates concatenation).

I_v = 0

I_d = 0xe166a6630 . UINT256(0x100)

At this point, we can use a neat feature of the Z3 solver: Not only can it tell us whether the formula is satisfiable, it can also produce a solution to the formula. The Mythril command line tool can do this automatically; simply run the security analysis and add the --verbose-report flag.

```
--------------------------------------------------------------------------------
$ myth -x assertions.sol --verbose-report

==== Exception state ====
Type: Informational
Contract: Assertions
Function name: assertion1(uint256)
PC address: 189

A reachable exception (opcode 0xfe) has been detected. This exception can
be caused by type errors, division by zero, out-of-bounds array access, or
assert violations. Note that assert() should be used only to check
invariants. Use require() for regular input-checking.


--------------------
In file: assertions.sol:4

assert(input * 4 < 1024)

--------------------

DEBUGGING INFORMATION:

The exception is triggered under the following conditions:

calldata_Assertions_4:
0000000000000000000000000000000000000000000000000000000000000200
calldata_Assertions_0:
e166a6630000000000000000000000000000000000000000000000000000000000000000
calldatasize_Assertions: 4
callvalue: 0

--------------------------------------------------------------------------------
```

Note the values calldata_Assertions_[number] in Mythril's output. These are the calldata values computed by the solver. Concatenating these values yields the transaction data that triggers the exception:

```
calldata =
0xe166a6630000000000000000000000000000000000000000000000000000000000000200
```

Granted, it doesn't take an Aristotle to come up with this result. But the solver can be applied to much more complex tasks, sometimes with surprising results. Let's look at an example that's a bit more challenging.

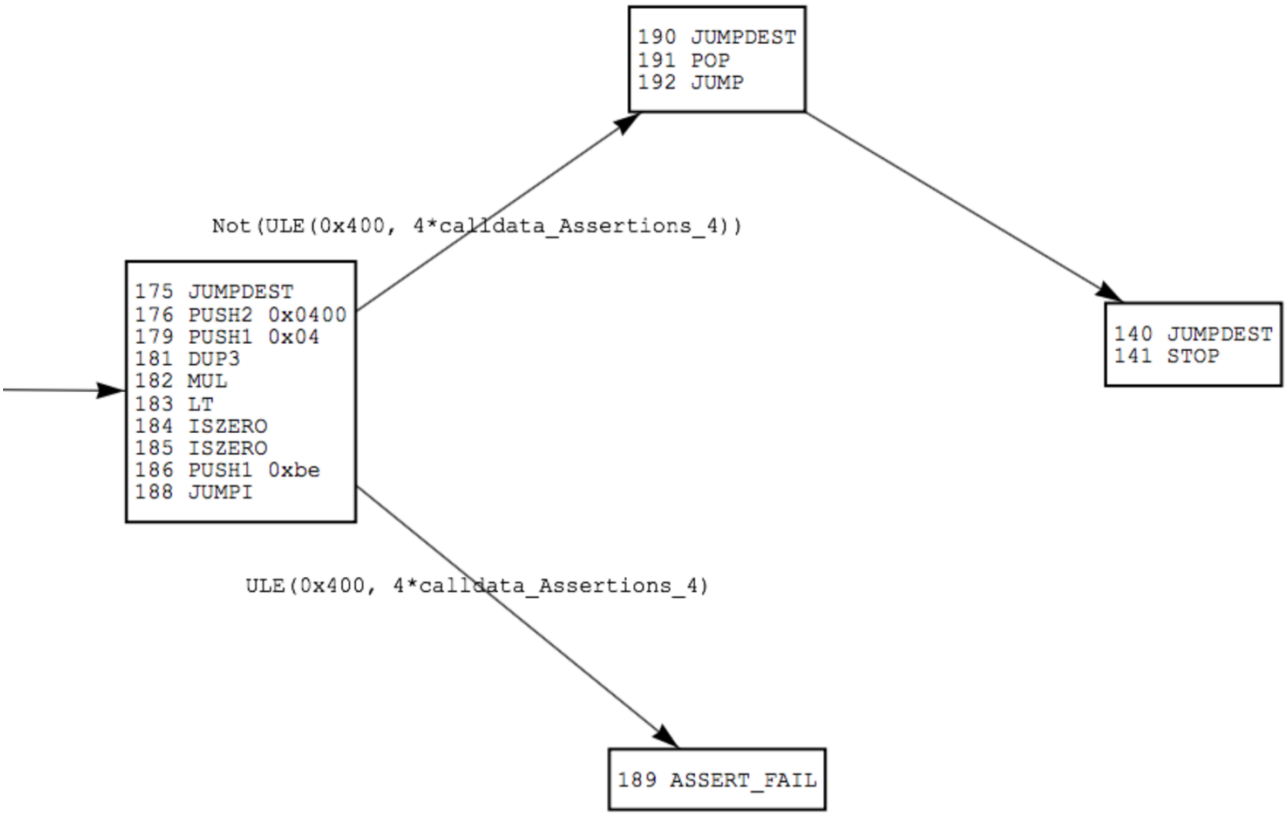


Figure 5: Control flow graph of the function assertion1()

UitwerpselenToken

In 2017, the first Underhanded Solidity Coding Contest [9] was held. The objective of the contest was to write harmless-looking Solidity code that conceals its purpose. One of many interesting submissions [10] was Doug Hoyte's MerdeToken [11], which was awarded a fourth place prize by the judges.

To demonstrate SMT Solving, I proudly present UitwerpselenToken, the spiritual successor to MerdeToken. UitwerpselenToken is a multi-user smart contract with functions for depositing and transferring Ether, and a bonus code management system. It supports two special accounts, *owner* and *manager*. The owner can withdraw any amount of Ether at any time, and the manager's task is managing the list of available bonus codes. Here's the full source code:

```
pragma solidity ^0.4.13;

// UitwerpselenToken: The multi-user wallet with a twist!
// Do NOT use this in production!

// UitwerpselenToken is based on MerdeToken by Doug Hoyte:
// https://github.com/Arachnid/uscc/tree/master/submissions-2017/doughoyte

contract UitwerpselenToken {
    address public owner;
    address public manager;

    function UitwerpselenToken(address _manager) {
        owner = msg.sender;
        manager = _manager;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    modifier onlyManager {
        require(msg.sender == manager);
        _;
    }

    mapping (address => uint) public balanceOf;
    uint public deposited;
```

```
    function deposit() payable {
        require(deposited + msg.value > deposited);
        require(balanceOf[msg.sender] + msg.value > balanceOf[msg.sender]);
        balanceOf[msg.sender] += msg.value;
        deposited += msg.value;
    }

    function balanceOf(address owner) constant returns (uint balance) {
        return balanceOf[owner];
    }

    function transfer(address to, uint value) {
        require(balanceOf[msg.sender] >= value);
        require(balanceOf[to] + value > balanceOf[to]);
        balanceOf[msg.sender] -= value;
        balanceOf[to] += value;
    }

    function withdraw(uint amount) onlyOwner {
        require(amount <= deposited);
        deposited -= amount;
        msg.sender.transfer(amount);
    }

    uint[] public bonusCodes;

    function pushBonusCode(uint code) onlyManager {
        bonusCodes.push(code);
    }

    function popBonusCode() onlyManager {
        require(bonusCodes.length >= 0);
        bonusCodes.length--;
    }

    function modifyBonusCode(uint index, uint update) onlyManager {
        require(index < bonusCodes.length);
        bonusCodes[index] = update;
    }
}
```

The code looks straightforward, and access privileges are clearly divided between the manager and owner roles. There is no obvious way for anyone besides the owner to withdraw Ether. Also, the owner address is set in the

constructor and there's no function that can change it. Let’s formally verify this to be on the safe side.

The first and most important step of formal verification is defining functional specifications, i.e., a list of provable statements about the code’s behavior. A formal statement about a given behavioral aspect of the code is called a "safety property". In the earlier assertion-based example, the safety properties were explicitly stated in the form of assert() calls. This time, we'll list general assumptions about the code and then translate them into verifiable formal statements.

Several assertions can be made about UitwerpselenToken. In this example, we'll have a closer look at two safety properties that describe the expected behavior with respect to Ether withdrawals and the contract owner:

1. Only the contract owner is authorized to withdraw Ether.
2. The owner address cannot be changed from its initial value (set in the constructor).

Let's now formalize this in the context of EVM bytecode execution. First, we need to determine how the state variable owner is represented on the bytecode level. In the EVM, storage is implemented as a key-value-store with 32-byte keys (aka "positions" or "slots") and 32-byte values. Statically sized variables are laid out contiguously in storage, starting from position 0. Because "address public owner" is the first state variable defined in UitwerpselenToken, it is represented by the value at storage position 0.

In yellow paper parlance, this storage position would be denoted by $\sigma_t[a]_s[o]$. We can now write the above safety properties as follows:

1. External calls with non-zero values are reachable if and only if $\sigma_t[a]_s[o] = I_s$.
2. There exists no state transition σ_t -> σ_{t+1} such that $\sigma_t[a]_s[o]$!= $\sigma_{t+1}[a]_s[o]$.

For the sake of simplicity, we’ll take it for granted that the code can be proven correct with respect to property 1. We’ll take a closer look at property 2, immutability of the owner state variable.

An important observation is that, according to the EVM specification, SSTORE is the only instruction capable of modifying account storage. We can therefore efficiently detect possible violations by looking for states in which $I_b[I_{pc}]$ = SSTORE, such that either mstate.stack[0] = 0 or mstate.stack[0] contains a symbolic variable x and the path formula P(x,I) is satisfiable given x = 0.

The following Python program implements the search using Mythril and Z3.

```
--------------------------------------------------------------------------
#!/usr/bin/env python

from laser.ethereum import svm
from mythril.ether.soliditycontract import SolidityContract
from mythril.analysis import solver
from mythril.exceptions import UnsatError
from z3 import *

address = "0x0000000000000000000000000000000000000000"

contract = SolidityContract("uitwerpselentoken.sol")

account = svm.Account(address, contract.disassembly)
accounts = {address: account}

laser = svm.LaserEVM(accounts)
laser.sym_exec(address)

for k in laser.nodes:

    node = laser.nodes[k]

    for state in node.states:

        if (state.get_current_instruction()['opcode'] == 'SSTORE'):

            proposition = node.constraints
            proposition.append(state.mstate.stack[-1] == 0)

            try:
                model = solver.get_model(proposition)

                print("Violation found!")

                for d in model.decls():
                    print("%s = 0x%x\n" % (d.name(), model[d].as_long()))

                codeinfo =
contract.get_source_info(state.get_current_instruction()['address'])

                print("%s\n%s\n" % (codeinfo.lineno, codeinfo.code))

            except UnsatError:
                pass

print("Analysis completed.")
--------------------------------------------------------------------------
```

Running the program produces the following output (I omitted some of the output because it filled multiple pages and some of the source mappings were off):

```
--------------------------------------------------------------------------------
$ python doorspoelen.py

VIOLATION FOUND:

In line 45:
balanceOf[msg.sender] -= value
(...)
keccac_146150163733090291820368483271628301965593254297 5_&
146150163733090291820368483271628301965593254297 5_&
caller = 0x0

VIOLATION FOUND:

In line 46:
balanceOf[to] += value

(...)
keccac_146150163733090291820368483271628301965593254297 5_&
146150163733090291820368483271628301965593254297 5_&
caller = 0x0
callvalue = 0x0


VIOLATION FOUND:

In line 68:
bonusCodes[index] = update

storage_4 =
0x75ca53043ea007e5c65182cbb028f60d717c000000000000000000000000000
storage_1 = 0x0
caller = 0x0
calldata_unknown_0 =
0xafdb35d700000000000000000000000000000000000000000000000000000000
calldatasize_unknown = 0x4
calldata_unknown_4 =
0x75ca53043ea007e5c65182cbb028f60d7179ff4b55739a3949b401801c942e65
callvalue = 0x0

(...)

VIOLATION FOUND:

In line 58:
```

```
bonusCodes.push(code)

storage_1 = 0x0
caller = 0x0
calldata_unknown_0 =
0xd456118a0000000000000000000000000000000000000000000000000000000000000000
calldatasize_unknown = 0x4
storage_4 =
0x75ca53043ea007e5c65182cbb028f60d7179ff4b55739a3949b401801c942e65
callvalue = 0x0

--------------------------------------------------------------------------------
```

Plot twist: The analysis outputs multiple violations! Perhaps the owner variable isn't immutable after all? Let's have a closer look at the results. Strangely, one group of findings points to the code lines that access the mapping balanceOf:

In line 32:

```
balanceOf[msg.sender] += msg.value
```

In line 43:

```
balanceOf[msg.sender] -= value
```

In line 44:

```
balanceOf[to] += value
```

The solver output hints at the possibility of writing to storage position 0 by supplying an address k so that keccak256(k . p) = 0. Unfortunately, the odds of finding this magic address are $1/2^{256}$. Those are pretty bad odds. You'd be about as likely to succeed at finding one specific atom out of all the atoms in the universe (and even if you were lucky, there might not be a solution at all because k is an uint256 with the leftmost 12 bits always zeroed out). We can safely disregard this as a false positive.

Another warning produced by our analysis tool:

```
--------------------------------------------------------------------------------
VIOLATION FOUND:

In line 58:
bonusCodes.push(code)

storage_4 =
0x75ca53043ea007e5c65182cbb028f60d7179ff4b55739a3949b401801c942e65
--------------------------------------------------------------------------------
```

Under certain conditions, pushing an element onto the array bonusCodes results in that element's being written to storage position 0.

The explanation is related to the method used to compute the storage position of the dynamic array's elements. Each array is assigned a storage slot at some position p. The storage slot itself contains the length of the array, and the storage address corresponding to an array element at some index is calculated as keccak256(p) + index (read [12] for a detailed explanation).

Once the array reaches a length x such that keccak256(p) + x = MAX_INT and the next element is pushed, the calculation overflows and the new element is stored at storage position 0.

Being the fourth variable declared in the program, the length of the bonusCodes array is stored at storage position 4, and elements are addressed as keccak256(4) + offset. Note that the solver provides the value for the "critical" length: storage_4 = 0x75(...). We can calculate this value ourselves:

```
x = 2 ** 256 - keccak256(4)
```

If the theory is correct, the result of this calculation should match the solver output.

```
--------------------------------------------------------------------------------
>>> from ethereum.utils import sha3
>>> hash = sha3((4).to_bytes(32, byteorder='big'))
>>> num = int.from_bytes(hash, byteorder='big')
>>> hex(2**256-num)
'0x75ca53043ea007e5c65182cbb028f60d7179ff4b55739a3949b401801c942e65'
--------------------------------------------------------------------------------
```

Great! Unfortunately, while this all sounds very nice in theory, nobody will ever exploit it in reality unless they somehow manage to push trillions of elements onto the array. Therefore, let's move on to the next result.

The solver shows a possible violation of the safety property at line 68 in the code:

```
--------------------------------------------------------------------------------
VIOLATION FOUND:

In line 68:
bonusCodes[index] = update

storage_4 =
0x75ca53043ea007e5c65182cbb028f60d717c0000000000000000000000000000
storage_1 = 0x0
```

```
caller = 0x0
calldata_unknown_0 =
0xafdb35d700000000000000000000000000000000000000000000000000000000
calldatasize_unknown = 0x4
calldata_unknown_4 =
0x75ca53043ea007e5c65182cbb028f60d7179ff4b55739a3949b401801c942e65
callvalue = 0x0
```

Once again, the dynamically sized array bonusCodes is involved, but this time the array element is addressed directly via the "index" function argument. For the same reasons discussed above, this causes a write to storage position 0 if index = 2 ** 256 - keccak256(4). Exploitation seems more realistic here since we can pass arbitrary uint256 values. Unfortunately, there is still a catch: Passing an index greater than the length of the array causes an exception. We therefore need a way to set the array length to a value greater than 2 ** 256 - keccak256(4).

Exploitation

If you haven't yet noticed anything weird in the function popBonusCode(), take another look:

```
    function popBonusCode() onlyManager {
        require(bonusCodes.length >= 0);
        bonusCodes.length--;
    }
```

In the sanity check, there's an off-by-one error that allows for underflowing the bonusCodes.length variable, causing it to assume the maximum unsigned integer value ($2^{256} - 1$). Addressing array indices without restrictions is now possible.

********** PRO TIP **********

Mythril comes with a module for detecting integer underflows (I've removed *over*flow detection for the time being because it caused too many false positives). You can use the -m argument to select specific security analysis modules:

```
$ myth -x uitwerpselentoken.sol -m integer s
==== Integer Underflow ====
Type: Warning
Contract: UitwerpselenToken
```

```
Function name: popBonusCode()
PC address: 1806
A possible integer underflow exists in the function popBonusCode().
The SUB instruction at address 1806 may result in a value < 0.
--------------------
In file: uitwerpselentoken.sol:63

bonusCodes.length--

**************************************************************************
```

The following is the exploit sequence for setting a new owner:

1. Underflow bonusCodes.length by calling popBonusCode().
2. Call modifyBonuscode(2 ** 256 - keccak256(2), new_owner_address).

We can test this out with a Truffle script:

```
------------------------------------------------------------------------------
module.exports = function(callback) {

    var UitwerpselenToken = arti-facts.require("./UitwerpselenToken.sol");
var manager = web3.eth.accounts[1];
    var token;

    UitwerpselenToken.deployed().then(function(instance) {
        token = instance;
        token.popBonusCode({from: manager});
    }).then(function() {
      return
token.modifyBonusCode("532780793507091663162804232028493225191901865910718511148743532101784727834 61",
"0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef", {from: manager});
    }).then(function() {
        return token.owner.call();
    }).then(function(new_owner) {
        console.log(new_owner);
    });
}
------------------------------------------------------------------------------
```

Executing the Truffle script produces the following output:

```
---------------------------------------------------------------------------
$ truffle exec test.js
Using network 'development'.

Owner before modifyBonusCode: 0x627306090abab3a6e1400e9345bc60c78a8bef57
Owner after modifyBonusCode: 0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
---------------------------------------------------------------------------
```

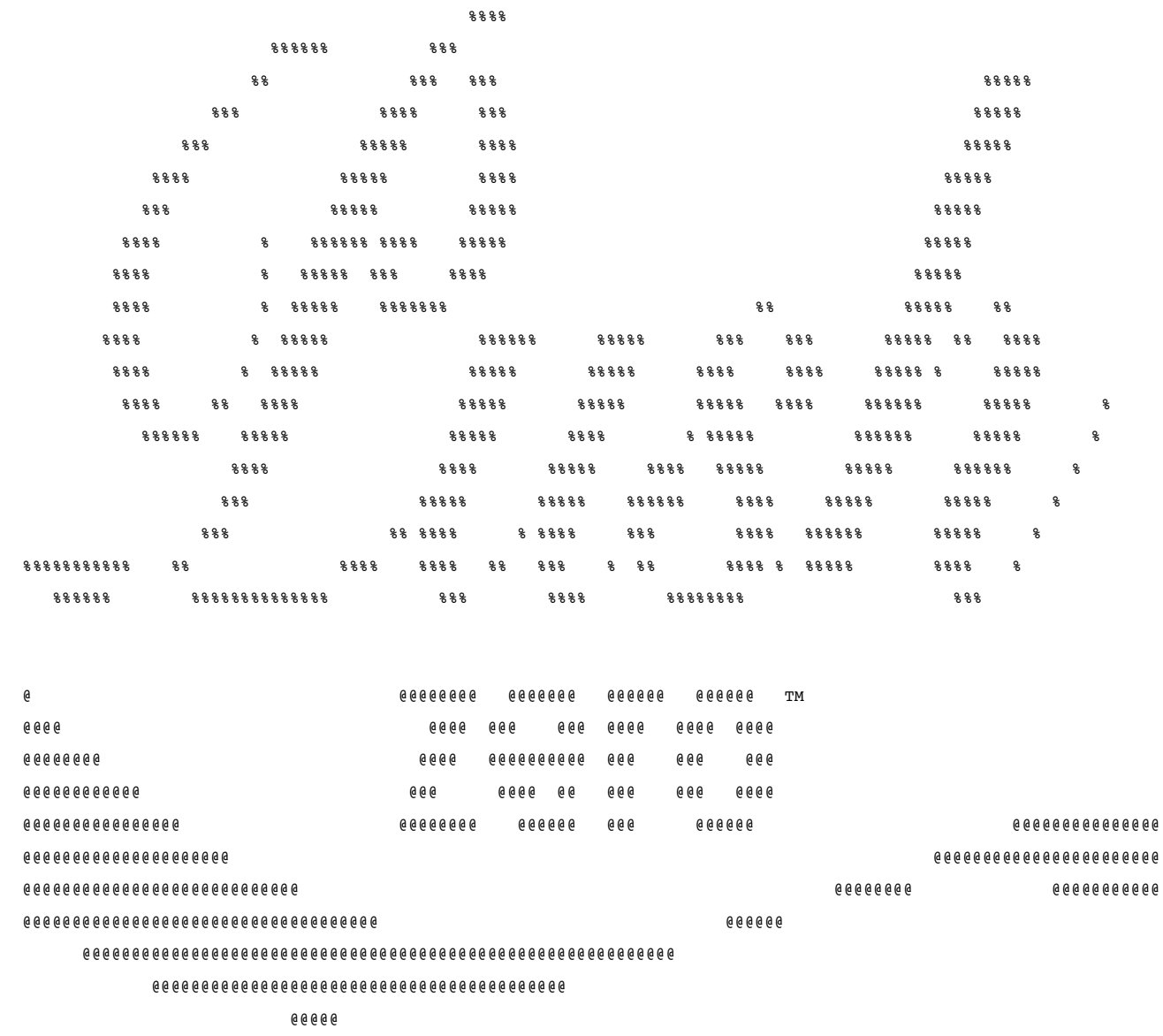
It worked :)

This vulnerability may seem unlikely to occur in practice, but it's not impossible. For example, explicitly decreasing array length is a pattern that Solidity developers use.

The main takeaway however is that formal analysis can detect bytecode-level issues that may not be obvious to an auditor inspecting the source code. Carefully defined safety properties and verification procedures should be a part of every auditor's toolbox.

********** PRO TIP **********

Whenever you discover a possible security issue (even if barely exploitable in practice), your first concern should be branding. Branding helps create awareness about the issue and earns you well-deserved attention. The branding exercise usually includes a name, a logo, and a website. I am therefore branding the above attack "push-into-zero," PUSHZERO for short, and proposing the following logo:



CTF Easy-Mode

The traditional use case for symbolic execution is binary cracking. For our final example, we'll use LASER to solve a CTF challenge. This isn't really practically applicable (how often does one need to crack smart contract DRM?) but it's a great way to understand the capabilities of symbolic execution and the Z3 Solver.

The following Ethereum CTF challenge was developed by Korantin Auguste for the Toulouse Hacking Convention, and it is printed with permission of the author. The source code is available on GitHub [13].

Someone sent me the bytecode of a smart contract that lives in the Ethereum blockchain. She told me, the smart contract is very simple: it contains a win() method, that anybody can call. This method takes a string as a parameter. If the string is the flag, it will call "selfdestruct(msg.sender)" to kill itself and send all its money to the sender. Can you find the flag?

Here is the bytecode of the contract (as deployed on the blockchain):

```
606060405260043610610041576000357c0100000000000000000000000000000000
00000000000000000000000000000900463ffffffff16806350f753bd146100465
75b600080fd5b34156100515760080fd5b6100736004803568ffffffffffff
ffffff19169060200190919050506100755565b005b600080602a91506000905
05b60178160ff1610156100cf57600810260ff168260ff16690100000000000000
00000000268ffffffffffffffffff19169060020a028318925060036007830020
191508080600101915050610081565b7f737461636b206d616368696e6573206
172652066756e2e0000000000000000008368ffffffffffffffffff191614156
1010c57818060010192505050b7f0631194a95069d7e012c19795d0c5c4ccd4af
1984e455700000000000000000008368ffffffffffffffffff191614156101595
73373ffffffffffffffffffffffffffffffffffffffff16ff5b7f3f6c6972687
4796d20726f202c65746e65796f2079725400000000000000000008368fffffff
fffffffffff191614156101965781806001019250505b5050505600a165627a7
a72305820afb3eeebc91132522b5075e270bcba664967c8ee60857ec0172ba60
eb4b264870029
```

Here is the prototype of the win function:

function win(bytes23 flag)

I know you are good at doing Cyber™ smart contracts over the blockchain in ze machine-learning cloud. Now is your time.

We start the analysis by creating a call graph and disassembling with Mythril.

```
$ myth --max-depth 64 -g ./ctf.html "6060604052[...]8260f"
$ myth -dc "6060604052[...]8260f"
(...)
269 PUSH32
0x0631194a95069d7e012c19795d0c5c4ccd4af1984e4557000000000000000000
302 DUP4
303 PUSH9 0xffffffffffffffff
313 NOT
314 AND
315 EQ
316 ISZERO
317 PUSH2 0x0159
320 JUMPI
321 CALLER
322 PUSH20 0xffffffffffffffffffffffffffffffffffffffff
343 AND
344 SUICIDE
(...)
```

In the assembly snippet, we find the SUICIDE instruction we wish to execute at PC address 344. At address 320 there's a conditional jump (JUMPI) that depends on the comparison of the input data with a hardcoded string. In the control flow graph (Figure 6), we can see an XOR decoder loop represented by two repeating basic blocks. These blocks are executed for a total of 23 times (note that loops are always unrolled in the state space, so a separate node is created for each iteration).

********** PRO TIP **********

For a more convenient way of reverse engineering smart contracts, look into these GUI-based smart contract disassemblers created by TrailOfBits:

- Ethersplay [14] is a graphical EVM disassembler capable of method recovery, dynamic jump computation, source code matching, and binary diffing. mEthersplay takes EVM bytecode as input and produces a binary file that can be analyzed in Binary Ninja.

- IDA-EVM [15] is a graphical EVM disassembler for IDA Pro that is capable of function recovery, dynamic jump computation, library signatures application, and binary diffing via BinDiff.

Another very useful reverse engineering tool is Porisity [16], a decompiler for EVM bytecode that was first introduced at DEFCon 25 [17].

****

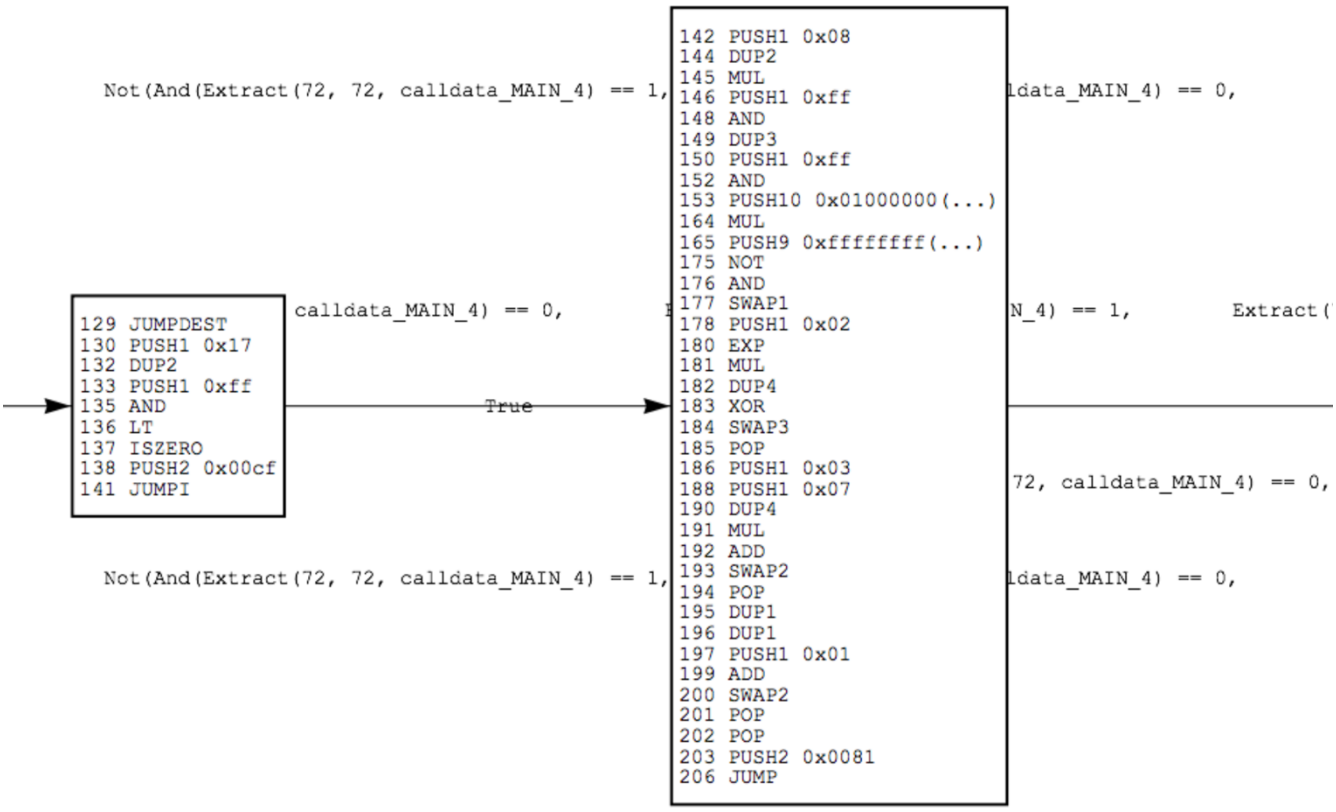


Figure 6: XOR decoder loop (the two blocks are repeated 23 times).

Thanks to symbolic execution, we don't really need to worry about what exactly the decoder does or what the hardcoded string contains. All we need to do is search for a state in which the SUICIDE instruction is executed and ask Z3 for a solution to the path formula of that state. We don't even have to write any extra code because Mythril happens to search for unprotected SUICIDE instructions by default. Adding the --verbose-reports flag causes Mythril to report the solution.

```
--------------------------------------------------------------------------------
$ myth -m suicide --max-depth 64 --verbose-report -xc "606060(…)0029"

==== Unchecked SUICIDE ====
Type: Warning
Contract: MAIN
Function name: _function_0x50f753bd
PC address: 344
The function _function_0x50f753bd executes the SUICIDE instruction. The
remaining Ether is sent to the caller's address.
```

```
It seems that this function can be called without restrictions.
--------------------

DEBUGGING INFORMATION:

SOLVER OUTPUT:
calldata_MAIN_4:
0x5448437b776f775f737563685f45564d5f736b696c6c7d000000000000000000
calldata_MAIN_0:
0x50f753bd0000000000000000000000000000000000000000000000000000000
calldatasize_MAIN: 0x4
callvalue: 0x0
--------------------------------------------------------------------------------
```

In the "debugging information" section, calldata_MAIN_4 shows the input needed to reach the SUICIDE instruction at PC address 344: These are the ASCII bytes of the flag we're looking for.

```
--------------------------------------------------------------------------------
>>> >>>
bytes.fromhex("5448437b776f775f737563685f45564d5f736b696c6c7d").decode("UTF
8")
'THC{wow_such_EVM_skill}'
--------------------------------------------------------------------------------
```

This was so easy that it almost feels like cheating! On the other hand, there aren't any rules in binary cracking, so let's not beat ourselves up too much about it.

Multi-Transactional Concolic Analysis

Up until now, we have analyzed abstract representations of single contract invocations. However, in some cases this is not sufficient for deciding whether a vulnerability is exploitable. For example, exploitability often depends on the concrete account state at the time of the attack (e.g., whether a particular variable has been initialized) and might require successive invocations of the contract (e.g., first underflow an integer variable, then address an array element). Nikolic´ et al. refer to this class of issues as *trace vulnerabilities* [18].

Concolic execution incorporates both *conc*rete and symb*olic* variables. Mythril's analysis does use concrete values when they're set explicitly during execution, but all state variables are initially symbolic. A possible improvement would be to initialize the world state with concrete

values either by running the constructor (for source code input) or by reading the referenced state variables from the blockchain.

Starting from the initial state, we can then symbolically execute the bytecode and save the set of possible final states (i.e., changed world states after normal termination). The final states are input for the next round of symbolic execution. The result is a tree of global states (Figure 7).

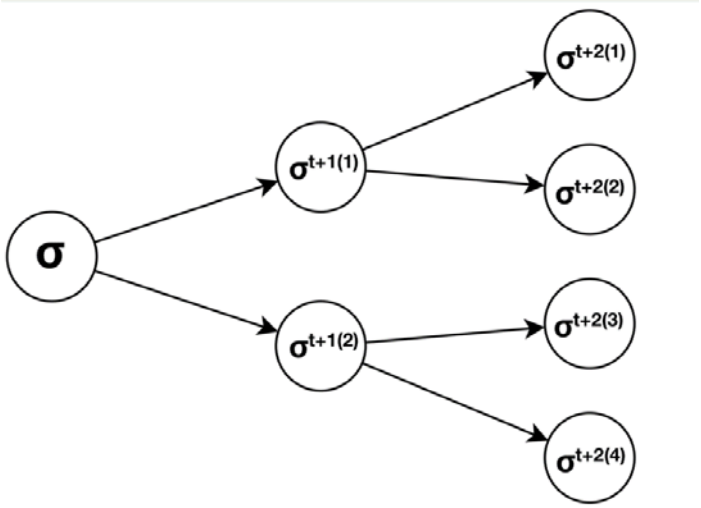


Figure 7: Tree of world states in a multi-transactional concolic execution

The lack of a multi-transactional model and concrete initial states in Mythril's analysis means that while the individual reasoning steps it performs are sound, a higher number of false positives is to be expected, and manual investigation is needed to verify the results. Manticore [19] and MAIAN [18] both support security analysis that considers multiple transaction.

Taking Profits

A lot has changed since 1996: VHS cassettes have been replaced by Netflix, Pluto is no longer classified as a planet, and mobile phones have become so fragile that they can't even survive a 0.9 meter drop [20]. Unfortunately however, some important technologies, such as lightsabers and secure programming languages, have failed to materialize.

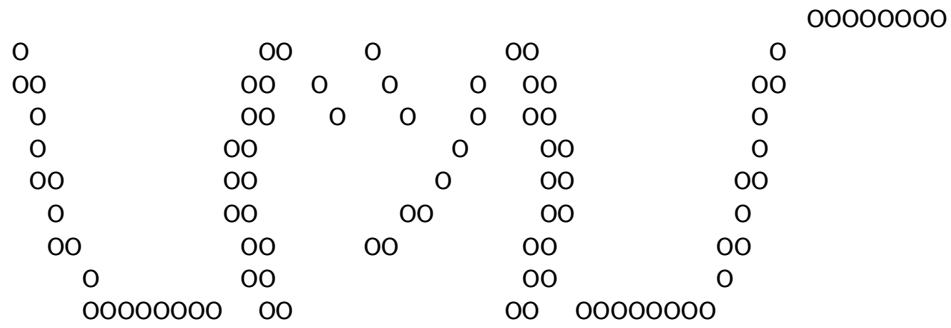
Even Solidity, the most popular language in the Ethereum world, is not as solid as the name seems to indicate. As Matt Kindy put it [21]:

"By trying to make smart contract writing as superficially accessible as possible, we've unwittingly assigned the critical task of designing once-deployable, unfamiliar patterned financial software to full stack devs who are used to building things that have completely incomparable degrees of adverse effects from failure."

Writing safe smart contracts requires knowledge about the subtleties of Solidity language constructs as well as a contextual understanding of how code executes on the blockchain. In the following section, we'll look into the causes of real-world vulnerabilities which, in some cases, were exploited for considerable profit (or caused significant losses).

Accidentally Born Killer

Profit: -$350,000,000 USD



On 6 November 2017, a new issue was created on the Parity GitHub repository [22]:

--- anyone can kill your contract - #6995 ---

devops199 opened this issue on Nov 6, 2017

I accidentally killed it.

Although the issue was dismissed at first, the killed contract turned out to be Parity's official multi-signature wallet library; all wallets depending on the library had become unusable. Even worse, users soon realized that all funds in the affected wallets were permanently frozen (to witness the gradual transition from disbelief to the realization that all funds were lost forever, read the discussion on GitHub [22]).

Even though the circumstances of the "hack" were almost comical, the consequences were anything but. Estimates of the amount destroyed range from $150 to $300 million USD [23], including the ICO proceedings of the Polkadot project [24]. Understandably, some of the affected fund-holders have since been pushing for a funds recovery process, creating ongoing controversy [25].

What happened? The transaction history documenting the murder can be viewed on Etherscan [26]. With the first transaction, devops199 called the initWallet() function to initialize the owners array:

```
--------------------------------------------------------------------------------
Function: initWallet(address[] _owners, uint256 _required, uint256
_daylimit)
MethodID: 0xe46dcfeb
[0]:0000000000000000000000000000000000000000000000000000000000000060
[1]:0000000000000000000000000000000000000000000000000000000000000000
[2]:0000000000000000000000000000000000000000000000000000000000000000
[3]:0000000000000000000000000000000000000000000000000000000000000001
[4]:000000000000000000000000ae7168deb525862f4fee37d987a971b385b96952
--------------------------------------------------------------------------------
```

Subsequently, devops199 (accidentally) sent another transaction, which executed the kill() function:

```
--------------------------------------------------------------------------------
Function: kill(address _to)
MethodID: 0xcbf0b0c0
[0]:000000000000000000000000ae7168deb525862f4fee37d987a971b385b96952
--------------------------------------------------------------------------------
```

You might rightly be wondering why this worked so easily. Surely there were checks in place to prevent random senders from calling the initWallet() function? As you can see in the source code, access to this function is indeed restricted with the only_uninitialized modifier:

```
--------------------------------------------------------------------------------
  // throw unless the contract is not yet initialized.
  modifier only_uninitialized { if (m_numOwners > 0) throw; _; }

  // constructor - just pass on the owner array to the multiowned and
  // the limit to daylimit
  function initWallet(address[] _owners, uint _required, uint _daylimit)
only_uninitialized {
    initDaylimit(_daylimit);
```

```
    initMultiowned(_owners, _required);
  }
```

This modifier protected the contract from being initialized more than once. The only problem was that WalletLibrary had never been initialized! After all, the WalletLibrary contract account wasn't supposed to be used as a wallet; it was intended solely as a code repository for other contracts, which would re-use its code via the delegatecall() function.

I'll revisit delegatecall() and its implications a bit later, but for now let's focus on the key problems in the devops199 incident.

Why did this happen? The root cause is Parity's developers' having implemented their library in a sloppy way. Solidity offers a library construct [27], a special type of contract that is assumed to be stateless. Parity didn't use it. Instead, they deployed a slightly modified version of a regular wallet that had its own state variables. This caused multiple problems, including the fact that the library instance itself could be "turned into a wallet" (a second vulnerability is detailed later in this paper).

Detection Strategy

Mythril currently has a simplistic module for detecting unprotected and weakly protected SUICIDE instructions. The basic concept of the module is to check the constraints on states in which the SUICIDE instruction is executed. If there is an unconstrained path, an alert is created. If there are one or more constraints on storage positions, Mythril checks the state space for potential writes to those storage positions. If such writes exist, an alert is generated.

The test successfully generates an alert when run with the Parity WalletLibrary:

```
--------------------------------------------------------------------------------
$ myth -xo markdown WalletLibrary.sol:WalletLibrary


Unchecked SUICIDE
==== Unchecked SUICIDE ====
Type: Warning
Contract: WalletLibrary
Function name: kill(address)
PC address: 6074
The function kill(address) executes the SUICIDE instruction. The Ether is
sent to an address provided as a function argument.

There is a check on storage index 21. This storage index can be written to
by calling the function 'initMultiowned(address[],uint256)'.
--------------------
```

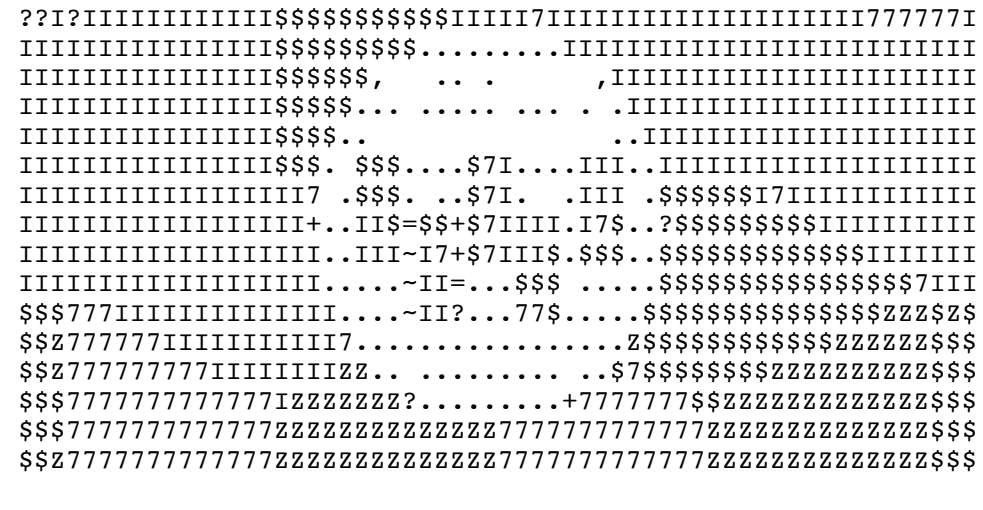
In file: WalletLibrary.sol:226

suicide(_to)
Note that the analysis doesn't take into account the concrete state of the contract, i.e., whether the account has been initialized. This is impossible to determine given only the source code because initialization happens only after the contract has been deployed to the blockchain. To eliminate false positives, the concrete value of m_numOwners should be taken into account when analyzing a deployed contract instance.

De-Constructed

*** Profit: A few bucks ***

In March 2016, user Polly Stripe colorfully announced a new Ponzi smart contract on the BitcoinTalk forums [28].



Hello! My name is Rubixi!

I'm a new & verified pyramid smart contract running on the Ethereum blockchain.

When you send me 1 ether, I will multiply the amount and send it back to your address when the balance is sufficient.

The intriguing premise: By paying at least 1 Ether into a literal pyramid scheme, users would gain a chance to multiply their money, assuming enough additional users subsequently joined the lower end of the pyramid. Naturally, investors flocked to take advantage of this amazing opportunity.

From every transaction, the contract would collect a small fee of up to ten percent, which could be withdrawn by its creator. This was only fair, given the considerable effort that had gone into writing the code. Unfortunately for Polly Stripe, it turned out that *anyone* could take ownership of the contract and withdraw the fees.

Here are the relevant parts of the Rubixi source code [29]:

```
--------------------------------------------------------------------------------
contract Rubixi {

        //Declare variables for storage critical to contract

        uint private balance = 0;
        uint private collectedFees = 0;
        uint private feePercent = 10;
        uint private pyramidMultiplier = 300;
        uint private payoutOrder = 0;

        address private creator;

        //Sets creator
        function DynamicPyramid() {
                creator = msg.sender;
        }

        modifier onlyowner {
                if (msg.sender == creator) _
        }

        /*** code omitted for brevity ***/

        //Fee functions for creator
        function collectAllFees() onlyowner {
                if (collectedFees == 0) throw;

                creator.send(collectedFees);


        function collectFeesInEther(uint _amt) onlyowner {
                _amt *= 1 ether;
                if (_amt > collectedFees) collectAllFees();

                if (collectedFees == 0) throw;

                creator.send(_amt);
                collectedFees -= _amt;
        }

        function collectPercentOfFees(uint _pcent) onlyowner {
```

```
                if (collectedFees == 0 || _pcent > 100) throw;

                uint feesToCollect = collectedFees / 100 * _pcent;
                creator.send(feesToCollect);
                collectedFees -= feesToCollect;
        }

        /*** code omitted for brevity ***/
}
```

In Solidity, one can define a constructor function that is executed once when the contract is created [30]. This is useful for initializing state variables during contract creation. The constructor code is not included in the run time code deployed on the blockchain. For a function to be recognized as the constructor, the function name must be identical to the contract name.

Polly Stripe had created Rubixi by copying and pasting code from another contract, DynamicPyramid, but she had failed to rename the constructor function accordingly. Therefore, instead of being run at contract creation time, the initialization code had ended up in the run time bytecode. This allowed random users to crown themselves owner (you can still "take over" the contract today).

Detection Strategy

Mythril's detection module in ether_send.py works similarly to the SUICIDE detection module. The idea is to check any function that sends Ether to a user-supplied address. The following is the algorithm:

For every state in which I_{pc} = CALL:

1. Determine whether the call value is greater than zero.
2. Check the target stack address.
3. For every storage constraint on the node containing the CALL, search for SSTORE instructions that may allow writing to the storage slot.
4. Attempt to satisfy the state's path formula.
5. Report a potential issue if the address is user-supplied, the path formula can be satisfied, and either there are no storage constraints or there are potential paths to writing the respective storage locations.

Running the analysis on the Rubixi source code generates the following result:

```
$ myth -x rubixi.sol
```

```
==== Ether send ====
Type: Warning
Contract: Rubixi
Function name: collectAllFees()
PC address: 1940
In the function 'collectAllFees()' a non-zero amount of Ether is sent to an
address taken from storage slot 5There is a check on storage index 5. This
storage slot can be written to by calling the function 'DynamicPyramid()'.

There is a check on storage index 9. This storage slot can be written to by
calling the function DynamicPyramid().
There is a check on storage index 10. This storage slot can be written to
by calling the function fallback().
--------------------
In file: rubixi.sol:75

creator.send(collectedFees)

--------------------

==== Ether send ====
Type: Warning
Contract: Rubixi
Function name: collectPercentOf dFees(uint256)
PC address: 1599
In the function 'collectPercentOfFees(uint256)' a non-zero amount of Ether
is sent to an address taken from storage slot. There is a check on storage
index 5. This storage slot can be written to by calling the function
'DynamicPyramid()'.

There is a check on storage index 6. This storage slot can be written to by
calling the function 'DynamicPyramid()'.
There is a check on storage index 7. This storage slot can be written to by
calling the function 'fallback'.
--------------------
In file: rubixi.sol:93

creator.send(feesToCollect)

------------------------------------------------------------------------------
```

It's a pretty simple check, but it's sufficient for detecting trivial flaws and discovering interesting contracts on-chain (although it isn't false-positive-proof).

********** FUN FACT **********

Browsing Etherscan is a fun pastime. There are still 4.00326 Ether left in the Rubixi contract, and the transaction history [31] shows many failed attempts to pull them out.

Is there still profit available? The source code shows that it is possible to withdraw amounts less than or equal to the value in the collectedFees state variable. This is a private variable, but in Ethereum there are no *truly* private variables. Knowing that the variable is in storage position one, we can use Mythril to extract the contents.

```
$ myth -ia 0xe82719202e5965Cf5D9B6673B7503a3b92DE20be --storage 1
1: 0x0000000000000000000000000000000000000000000000000000000000000000
```

The value of collectedFees is zero, so withdrawing the remaining Ether is impossible. The remaining four Ether will be paid out to players once the game resumes and the total balance reaches approximately 45 Ether (this will never happen, so the funds are stuck forever).

****

********** NMAP FOR THE BLOCKCHAIN **********

What fun is detecting vulnerabilities if you can't detect them in real smart contracts deployed on the blockchain? Mythril can retrieve contract bytecode from the Ethereum network. The easiest way to do this is using the built-in INFURA support (this works out-of-the-box without setting up your own node). To analyze the original Rubixi smart contract instance on the mainnet, run:

```
$ myth -xia 0xe82719202e5965Cf5D9B6673B7503a3b92DE20be
```

There are built-in IPC and RPC presets for various setups. You can also build a local contract database (or query the leveldb of a local go-ethereum installation) to search for interesting functions and opcode patterns. Setting up a local node is recommended to use this feature, as downloading large amounts of contract data from INFURA would be very slow. See the Mythril README [2] for more details.

****

DAOsaster

*** Profit: It's complicated ***

On 30 April 2016, the DAO was launched on Ethereum Block 1428757. It was an exciting concept: a decentralized autonomous organization governed by a smart contract on the Ethereum blockchain and managed according to the votes of its investors. The DAO token sale was a resounding success, raising fourteen percent of the Ether in circulation. At the time, the funds raised were equivalent to $34 million USD [32]. It was the most successful crowdfunding event in history.

The actual operation, however, was less successful, unless you consider being hacked and drained of all funds a success. In June 2016, an attacker withdrew 3.6 million Ether from the DAO's smart contracts, the world's first large smart contract hack. Eventually, parts of the Ethereum community decided to hard-fork the Ethereum blockchain to recover the lost funds. However, not everyone was happy with this decision, and the original chain lived on as Ethereum Classic [33] [34].

The exploited vulnerability was a re-entrancy bug (called "recursive send bug" at the time) in the splitDAO() function. The relevant parts of the source code are shown below.

```
--------------------------------------------------------------------------
function splitDAO(
    uint _proposalID,
    address _newCurator
) noEther onlyTokenholders returns (bool _success) {

    /*** Some checks & create new DAO (elaborate computations removed) ***/

    // Move ether and assign new Tokens
    uint fundsToBeMoved =
        (balances[msg.sender] * p.splitData[0].splitBalance) /
        p.splitData[0].totalSupply;
    if
(p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sender)
== false)
    throw;

    /*** More elaborate computations removed ***/

    // Burn DAO Tokens
    Transfer(msg.sender, 0, balances[msg.sender]);
    withdrawRewardFor(msg.sender); // be nice, and get his rewards
```

```
    totalSupply -= balances[msg.sender];
    balances[msg.sender] = 0;
    paidOut[msg.sender] = 0;
    return true;
}
```

In summary, the splitDAO() function creates a "child DAO" contract and transfers tokens to the new DAO. The number of transferred tokens is calculated according to the user's balance. Near the end of the function, a reward is sent to the caller through withdrawRewardFor(), which in turn calls the payOut() function.

```
function payOut(address _recipient, uint _amount) returns (bool) {
    if (msg.sender != owner || msg.value > 0 || (payOwnerOnly && _recipient
!= owner))
        throw;
    if (_recipient.call.value(_amount)()) {
        PayOut(_recipient, _amount);
        return true;
    } else {
        return false;
    }
}
```

Note the use of the low-level call statement in payOut():

```
_recipient.call.value(_amount)()
```

The first issue here is that _recipient is a user-supplied address and could point to a malicious smart contract. Because there's no call data sent with the message call, the code in the target contract's fallback function will be executed.

The DAO attacker set up a malicious contract with a fallback function that called back into splitDAO(), starting a recursive loop. Note that the state variables, including the caller's balance, are updated only *after* the re-entrant call. This means that even though tokens are withdrawn at every iteration of the loop, the attacker's balance is never set to zero.

The actual hack was a bit more involved: The attacker exploited multiple flaws to amplify the attack's effectiveness. Phil Daian has examined the attack in detail in his two-part write-up [35] [36].

Detection Strategy

The code of the original DAO [37] is pretty convoluted, so I'll be working with a simple example. The following code is from level 10 of Zeppelin's Ethernaut challenges [38].

```
pragma solidity ^0.4.18;

/*
This is level 10 of the Zeppelin Ethernaut challenge.
The original code is available at:
https://ethernaut.zeppelin.solutions/level/0xf70706db003e94cfe4b5e27ffd891d
5c81b39488
*/

contract Reentrance {

  mapping(address => uint) public balances;

  function donate(address _to) public payable {
    balances[_to] += msg.value;
  }

  function balanceOf(address _who) public constant returns (uint balance) {
    return balances[_who];
  }

  function withdraw(uint _amount) public {
    if(balances[msg.sender] >= _amount) {
      if(msg.sender.call.value(_amount)()) {
        _amount;
      }
      balances[msg.sender] -= _amount;
    }
  }

  function() payable {}
}
```

The following is Mythril's strategy for detecting potential re-entrancy bugs:

1. Detect all message calls to user-supplied addresses that also forward gas. Note that Solidity's send() and transfer() functions set call gas to only 2,300; this setting prevents re-entrancy attacks.
2. If an external call to an untrusted address is detected, analyze the control flow graph for possible state changes that occur after the call returns. Generate a warning if a state change is detected.

Running Mythril on the example contract produces the following output:

```
--------------------------------------------------------------------------------
$ myth -mexternal_calls -x reentrance.sol
==== Message call to external contract ====
Type: Warning
Contract: Reentrance
Function name: withdraw(uint256)
PC address: 552
This contract executes a message call to the address of the transaction
sender. Generally, it is not recommended to call user-supplied adresses
using Solidity's call() construct. Note that attackers might leverage
reentrancy attacks to exploit race conditions or manipulate this contract's
state.
--------------------
In file: reentrance.sol:23

msg.sender.call.value(_amount)()

--------------------

==== State change after external call ====
Type: Warning
Contract: Reentrance
Function name: withdraw(uint256)
PC address: 632
The contract account state is changed after an external call. Consider that
the called contract could re-enter the function before this state change
takes place. This can lead to business logic vulnerabilities.
--------------------
In file: reentrance.sol:26

balances[msg.sender] -= _amount
--------------------------------------------------------------------------------
```

Running Mythril with the address of the DAO does detect eight instances of state changes after external calls. However, Mythril has trouble resolving the function names (I haven't had the time to investigate the results in detail).

```
--------------------------------------------------------------------------------
```

```
$ myth --max-depth 128 -mexternal_calls -xila
0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413
==== Message call to external contract ====
Type: Informational
Contract: 0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413
Function name: fallback
PC address: 2761
This contract executes a message call to another contract. Make sure that
the called contract is trusted and does not execute user-supplied code.
--------------------

==== State change after external call ====
Type: Warning
Contract: 0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413
Function name: fallback
PC address: 75
The contract account state is changed after an external call. Consider that
the called contract could re-enter the function before this state change
takes place. This can lead to business logic vulnerabilities.
--------------------

==== State change after external call ====
Type: Warning
Contract: 0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413
Function name: fallback
PC address: 2798
The contract account state is changed after an external call. Consider that
the called contract could re-enter the function before this state change
takes place. This can lead to business logic vulnerabilities.
--------------------

(...)
------------------------------------------------------------------------------
```

Delegation to Hell

*** Profit: 153,037 ETH ***

One fateful evening in November 2017, an individual known only by the codename "Mitch Brenner" went on seemingly normal Tinder date [39]. The evening started well but soon turned into a creep fest. Like most of us would, Mitch decided to return home and spend the rest of the night browsing transactions on Etherscan. Randomly skimming through the code of Parity's multisig wallet, he noticed something weird. In his own words:

"I remember finding it funny how Gav/Nicolas wrote that assembly piece there to call an internal method (initWallet) in the wallet library. Solidity is a shit language, so I wasn't all that surprised. But, I mean, really?"

Mitch whipped up a test setup in his private net and soon confirmed that "whomever has money in such wallet is possibly ducked." Naturally, the next question that came to his mind was how to exploit the flaw to steal as much Ether as humanly possible. By searching a memory-cached copy of the blockchain for instances of WalletLibrary and its dependents, he found a nice list of wallets and balances. Ultimately, he managed to empty three large wallets for a total of 153,037 ETH (see Mitch's original Medium post for the complete story [39]).

Let's look at the details of the attack. Remember the accidental suicide exploit? The function exploited by Mitch was the same function that later led to WalletLibrary's untimely demise. However, in the version Mitch attacked, the initWallet() function didn't yet have the only_uninitialized modifier [40].

```
--------------------------------------------------------------------------------
// constructor - just pass on the owner array to the multiowned and
// the limit to daylimit
function initWallet(address[] _owners, uint _required, uint _daylimit) {
  initMultiowned(_owners, _required);
  initDaylimit(_daylimit);
--------------------------------------------------------------------------------
```

Mitch did not target Walletlibrary directly. Instead, he made use of the fact that the initWallet() function was exposed in wallets that re-used the library's code through a delegatecall proxy. In the delegatecall proxy pattern, a contract forwards the received call data in its fallback function.

```
--------------------------------------------------------------------------------
// gets called when no other function matches
function() payable {
  // just being sent some cash?
  if (msg.value > 0)
    Deposit(msg.sender, msg.value);
  else if (msg.data.length > 0)
    _walletLibrary.delegatecall(msg.data);
}
--------------------------------------------------------------------------------
```

In principle, the root cause of the flaw was weakly protected functionality, as in the Rubixi and accidental suicide exploits. In this

case, however, the flaw involved interaction between multiple contracts. We therefore need to include those interactions in our analysis.

Modeling Message Calls

Call semantics are one of the most dazzling aspects of the EVM.

The first important thing is that *all* interactions between contract accounts are implemented as message calls: Sending Ether with the send(), transfer(), and call().value() Solidity functions compiles to a CALL opcode, just like explicitly calling a function in another contract does. Conversely, internal function calls do *not* compile into CALL instructions but use the JMP instruction instead.

In the context of the EVM, the word "call" thus refers to a message to and from a contract, not necessarily to a function call.

Depending on the context, there are many ways that message calls can be invoked. The methods differ by the following details:

- the opcode used (CALL, CALLCODE, DELEGATECALL, STATICCALL, etc.)
- whether call data is passed along
- the call value (amount of Ether transferred)
- the amount of gas sent with the call

Message calls always return a Boolean value: "True" if execution of the call terminated normally and "False" if execution terminated with an exception. For some Solidity constructs, the compiler inserts code to check the return value and propagate exceptions to the calling contract, but in low-level calls, execution continues even if the call fails.

It's fair to say that a developer from a JavaScript background probably won't expect this kind of complexity from something as mundane as a function call. Indeed, misunderstandings of message call semantics are a main source of vulnerabilities. For more information on using calls securely, I recommend reading the External Calls section of ConsenSys' Smart Contract Best Practices [41].

LASER can simulate different types of message calls with symbolic or concrete call data. To understand the difference between a regular call and a delegatecall, let's have a look at how these two call variants are represented in the state space.

Explicit external function calls translate into CALL instructions. Here is an example:

```
contract Callee {
    function theFunction() payable {
    }
}

contract Caller {

    address public callee_address;

    function Caller(address addr) {
        callee_address = addr;
    }

    function whatever() public {
        Callee(callee_address).theFunction();
    }
}
```

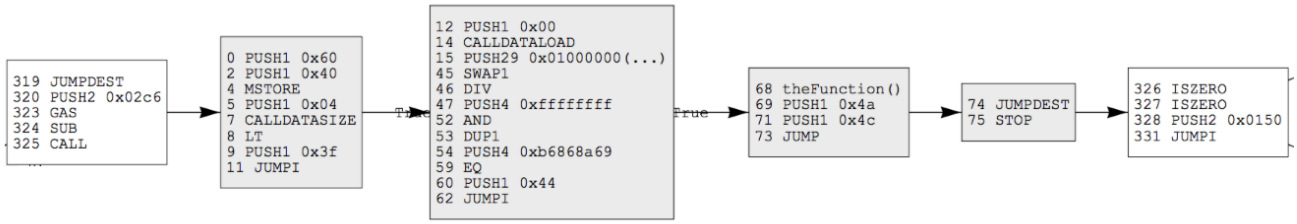


Figure 8: Control flow graph for a regular inter-contract message call

Note the CALL instruction executed at address 325, after which execution resumes at the beginning of the callee contract's bytecode (Figure 8). Because the call data contains the concrete function signature hash for theFunction(), the state space in the called contract is restricted to states of that function. After the callee terminates, execution resumes with the instruction after the CALL.

With the regular CALL instruction, the execution context switches to the callee account, and any state modifications (e.g., storage) affect the callee.

Let's now compare this with the abstract representation of a delegatecall proxy as it is found in the Parity WalletLibrary. The following contract implements a basic delegatecall proxy (its control flow graph is shown in Figure 9).

```
---------------------------------------------------------------------------
contract Delegate {

  function func() payable {
  }
}

contract Delegation {

  address delegate;

  function Delegation(address _delegateAddress) {
    delegate = Delegate(_delegateAddress);
  }

  function() payable {
    delegate.delegatecall(msg.data);
  }
}
---------------------------------------------------------------------------
```

In this case, the call value is symbolic, so execution of the delegate can take arbitrary paths. This results in a more complicated state space. Note that Mythril currently "summarizes" function returns from the delegate so that each return draws an edge to the same node in the caller (the rightmost node in Figure 9). I originally implemented it that way to enhance performance, but it causes inaccuracies, so I'll need to revise it eventually.

```
******************************** PRO TIP *********************************
Mythril has a built-in dynamic loader that will automatically load
dependencies from the blockchain when a CALL or DELEGATECALL opcode is
encountered. The dynamic loader is activated with the -l flag, for example


$ myth --ganache -lxa 0x471c92f915ae766c4964eedc300e5b8ff41e443c
**************************************************************************
```

In a delegatecall, msg.sender and msg.value retain their original values and the code at the target address is executed in the context of the *calling* contract's account. This means that storage operations in the delegate's code access the account storage of the calling account.

There is no semantic link between state variables in the caller. If the delegate accesses a state variable's storage position 0, the delegate will access whichever state variable happens to be mapped to the same position in the caller account.

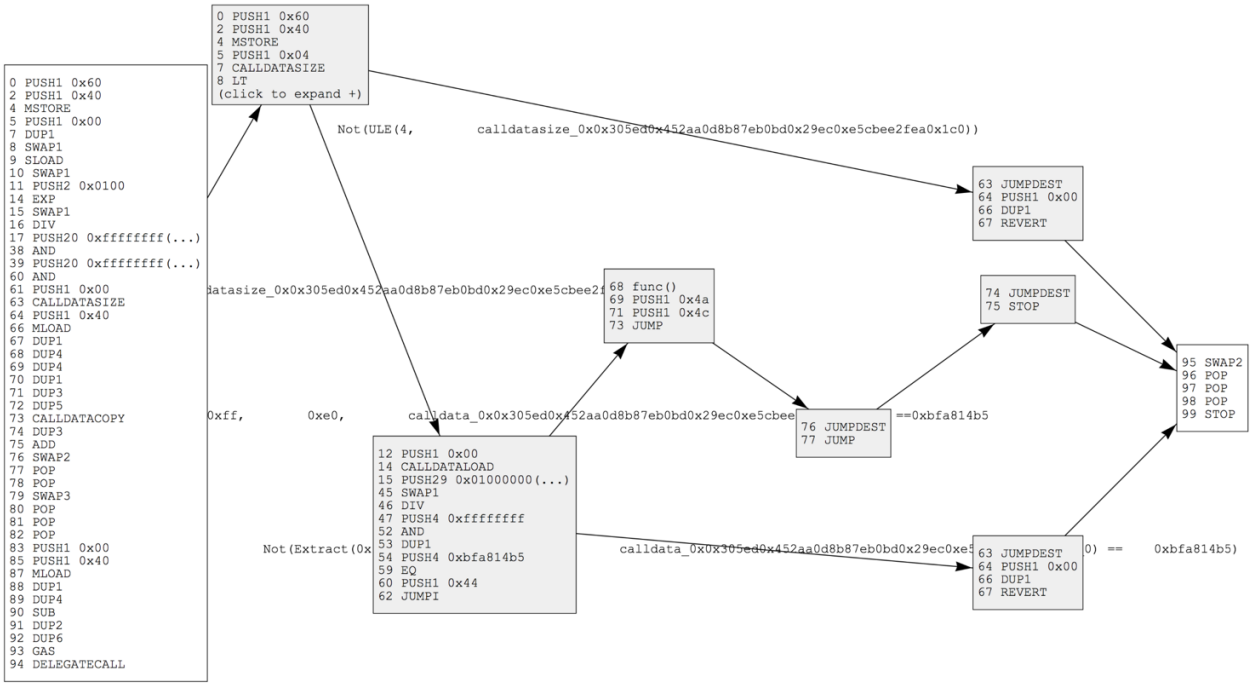


Figure 9: Control flow graph for a delegatecall proxy

Let's revisit the Parity wallet scenario. How did the Parity guys make sure that WalletLibrary's code had access to the Wallet's state variables? The solution was to include the same state variables (in the same order) in the source code of both the Wallet and the WalletLibrary:

```
  // FIELDS
  address constant _walletLibrary =
0xcafecafecafecafecafecafecafecafecafecafe;

  // the number of owners that must confirm the same operation before it is
run.
  uint public m_required;
  // pointer used to find a free slot in m_owners
  uint public m_numOwners;

  uint public m_dailyLimit;
  uint public m_spentToday;
  uint public m_lastDay;

  // list of owners
  uint[256] m_owners;
```

That way, the variable m_owners map to the dynamically sized array at storage position 7 in both contracts. If this seems like a rather hackish thing to do, that's because it is. It would have been better to implement a stateless library contract, as described in the Solidity documentation [27].

Detection Strategy

Once we have added the capability to execute message calls, the same "unprotected Ether send" algorithm that detects the Rubixi issue should work for this Parity issue. When Mythril encounters the delegatecall proxy, it retrieves WalletLibrary from the blockchain and creates a state space for both contracts.

Here is the analysis' result for the Parity wallet instance deployed on Ganache:

```
--------------------------------------------------------------------------------
myth --ganache -xla 0x8adc0ea8e50ca7fe84ba0a39b74442ff9a2d4afd


==== CALLDATA forwarded with delegatecall() ====
Type: Informational
Contract: 0x8adc0ea8e50ca7fe84ba0a39b74442ff9a2d4afd
Function name: fallback
PC address: 367

This contract forwards its calldata via delegatecall in its fallback
function. This means that any function in the called contract can be
executed. Note that the callee contract will have access to the storage of
the calling contract.
--------------------
==== Ether send ====
Type: Warning
Contract: 0x8adc0ea8e50ca7fe84ba0a39b74442ff9a2d4afd
Function name: execute(address,uint256,bytes)
PC address: 4165

In the function 'execute(address,uint256,bytes)' a non-zero amount of Ether
is sent to an address taken from function arguments.

There is a check on storage index 26. This storage slot can be written to
by calling the function 'initMultiowned(address[],uint256)'. There is a
check on storage index 29. This storage slot can be written to by calling
the function 'initMultiowned(address[],uint256)'.
```

Summary and Outlook

If you've been in software security for a while, you know that it's a hopeless field: Twenty-two years after the original "Smashing the Stack," we still haven't seen a piece of bug-free software. Where there's complexity and margin for human error, errors will be made.

When it comes to Ethereum smart contracts, I believe that we *do* have a shot at getting things right. Hoping that formally verifiable languages will replace Solidity anytime soon might be overly optimistic, but even smart contracts written in Solidity can be designed to be simple and verifiable. It is possible to create a world of secure smart contracts that users can trust.

Verified libraries and code patterns (such as those developed by OpenZeppelin [42]) are a great step in this direction. Security best practices [41] need to be translated into tools that support developers from the earliest stages of development onwards. We must also establish certification infrastructure: Every contract deployed on the mainnet should be required to show proof of secure SDLC and audit.

Security research is interesting and fun, but it's not very beneficial if the results don't trickle down into the real world. To make an impact, we need to create tools that are useful to developers and end users. Therefore, my next focus will be making Mythril more accessible via IDE plugins that work out of the box, continuous integration tools, comprehensive reporting, and a lower false positive rate. For ideas, feedback, or questions, create an issue in the Mythril GitHub repo [2] or join the Mythril Gitter chat [5].

References

[1] Aleph One, "Smashing the Stack for Fun and Profit (Phrack #49)," 1996. [Online]. Available: http://phrack.org/issues/49/14.html

[2] ConsenSys, "Mythril," [Online]. Available: https://github.com/ConsenSys/mythril

[3] B. Mueller, "LASER-Ethereum," [Online]. Available: https://github.com/b-mueller/laser-ethereum

[4] B. Mueller, "Smashing Smart Contracts Supplemental Materials," [Online]. Available: https://github.com/b-mueller/smashing-smart-contracts

[5] "Mythril Gitter," [Online]. Available: https://gitter.im/ConsenSys/mythril

[6] G. Wood, "Ethereum Yellow Paper, EIP-150 Revision," [Online]. Available: http://gavwood.com/paper.pdf

[7] M. Dameron, "Ethereum Beige Paper," [Online]. Available: https://github.com/chronaeon/beigepaper

[8] V. Buterin, "Ethereum White Paper," [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[9] "Underhanded Solidity Coding Contest," [Online]. Available: http://u.solidity.cc

[10] "Submissions to the USCC 2017 on Github," [Online]. Available: https://github.com/Arachnid/uscc/tree/master/submissions-2017/

[11] D. Hoyte, "MerdeToken," [Online]. Available: https://github.com/Arachnid/uscc/tree/master/submissions-2017/doughoyte

[12] H. Yeh, "Diving Into The Ethereum VM Part 3 — The Hidden Costs of Arrays," [Online]. Available: https://medium.com/@hayeah/diving-into-the-ethereum-vm-the-hidden-costs-of-arrays-28e119f04a9b

[13] K. Auguste, "THC CTF 2018 - Reverse of an ethereum smart contract on Github," [Online]. Available: https://github.com/ToulouseHackingConvention/reverse-palkeo-ethereum

[14] TrailOfBits, "Ethersplay," [Online]. Available: https://github.com/trailofbits/ethersplay

[15] TrailOfBits, "IDA-EVM," [Online]. Available: https://github.com/trailofbits/ida-evm

[16] Comae Technologies, "Porosity: Decompiler and Security Analysis tool for Blockchain-based Ethereum Smart-Contracts," [Online]. Available: https://github.com/comaeio/porosity

[17] M. Suiche, "Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode," 7 July 2017. [Online]. Available: https://www.comae.io/reports/dc25-msuiche-Porosity-Decompiling-Ethereum-Smart-Contracts-wp.pdf

[18] I. Nikolic´, A. Kolluri, I. Sergey, P. Saxena and A. Hobor, "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale," 14 March 2018. [Online]. Available: https://arxiv.org/pdf/1802.06038.pdf

[19] TrailofBits, "Manticore Symbolic Analysis Tool," [Online]. Available: https://github.com/trailofbits/manticore

[20] V. Orellana, "The iPhone X cracked on the first drop," [Online]. Available: https://www.cnet.com/news/apple-iphone-x-drop-test/

[21] M. Kindy, "For god's sake, can't we fix Solidity?," [Online]. Available: https://medium.com/topl-blog/for-gods-sake-cant-we-fix-solidity-9bc7184e2683

[22] devops199, "Anyone can kill your contract," 6 November 2017. [Online]. Available: https://github.com/paritytech/parity/issues/6995

[23] "ICO Funds Among Millions Frozen In Parity Wallets," Coindesk, 7 November 2017. [Online]. Available: https://www.coindesk.com/ico-funds-among-millions-frozen-parity-wallets/

[24] "Web3 Multi-Sig Wallet Update on Medium," [Online]. Available: https://medium.com/web3foundation/web-3-multi-sig-wallet-update-245d30df0fb3

[25] D. Phifer, "EIP 867 Ethereum Recovery Proposals (ERPs)," 2 February 2018. [Online]. Available: https://github.com/ethereum/EIPs/pull/867

[26] "Parity WalletLibrary on Etherscan," [Online]. Available: https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4

[27] "Solidity Documentation: Libraries," [Online]. Available: http://solidity.readthedocs.io/en/v0.4.21/contracts.html#libraries

[28] "Rubixi on the Bitcointalk Forums," [Online]. Available: https://bitcointalk.org/index.php?topic=1400536.0

[29] "Rubixi smart contract on Etherscan," [Online]. Available: https://etherscan.io/address/0xe82719202e5965Cf5D9B6673B7503a3b92DE20be

[30] "Solidity Documentation: Creating Contracts," [Online]. Available: http://solidity.readthedocs.io/en/develop/contracts.html#creating-contracts

[31] "Rubixi Transaction History on Etherscan," [Online]. Available: https://etherscan.io/address/0xe82719202e5965Cf5D9B6673B7503a3b92DE20be#transactions

[32] "Wikipedia Article: The DAO," [Online]. Available: https://en.wikipedia.org/wiki/The_DAO_(organization)

[33] D. Siegel, "Understanding the DAO Hack," 25 June 2016. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists/

[34] A. Madeira, "The DAO, The Hack, The Soft Fork and The Hard Fork," 20 March 2018. [Online]. Available: https://www.cryptocompare.com/coins/guides/the-dao-the-hack-the-soft-fork-and-the-hard-fork/

[35] P. Daian, "Analysis of the DAO exploit," 18 June 2016. [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[36] P. Daian, "Chasing the DAO Attacker's Wake," 19 June 2016. [Online]. Available: https://pdaian.com/blog/chasing-the-dao-attackers-wake/

[37] "TheDAO (Etherscan)," [Online]. Available: https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413

[38] Z. Solutions, "The Ethernaut Level 10," [Online]. Available: https://ethernaut.zeppelin.solutions/level/0xf70706db003e94cfe4b5e27ffd891d5c81b39488

[39] M. Brenner, "How I Snatched 153,037 ETH After A Bad Tinder Date on Medium," 13 September 2017. [Online]. Available: https://medium.com/@rtaylor30/how-i-snatched-your-153-037-eth-after-a-bad-tinder-date-d1d84422a50b

[40] "Parity Walletlibrary on Etherscan," [Online]. Available: https://etherscan.io/address/0x4f2875f631f4fc66b8e051defba0c9f9106d7d5a#code

[41] ConsenSys, "Smart Contract Best Practices," [Online]. Available: https://consensys.github.io/smart-contract-best-practices/recommendations/#external-calls

[42] OpenZeppelin, "Zeppelin Solidity," [Online]. Available: https://github.com/OpenZeppelin/zeppelin-solidity