Symbolic Searcher: Encode MEV Hackathon 2022

Amy Chou
amyachou@gmail.com

Bill Zhang
billzhang1998@gmail.com

Abstract

The detection and quantification of long-tail MEV is difficult and thus sparsely researched. By leveraging symbolic execution techniques, we develop a framework for evaluating certain long-tail MEV opportunities, and construct a symbolic execution searcher to algorithmically identify long-tail MEV opportunities. As a proof of concept, we apply this symbolic searcher on the canonical "Ethereum is a Dark Forest" MEV example to successfully identify the long-tail MEV present, thus demonstrating its potential as both a research tool for quantifying historical long-tail MEV, and as the prototype for a potentially profitable MEV searcher.

1 Introduction

The detection and quantification of MEV has become an important topic, with potentially billions of dollars in extracted value at play. However, research is quite difficult and thus sparse, especially regarding long-tail MEV. By leveraging symbolic execution techniques, we introduce an MEV classifier that algorithmically recognize historical long-tail MEV opportunities. In Section 2, we provide background on existing MEV quantification techniques, symbolic execution and its nascent usage towards MEV, and the canonical "Ethereum is a Dark Forest" MEV opportunity which will be used as an example throughout the paper. In Section 3, we outline a toy example that uses symbolic execution to identify MEV, then in Section 4 we relax the assumptions from the example to form a generalized proof-of-concept symbolic searcher. Section 5 uses the "Ethereum is a Dark Forest" MEV opportunity to demonstrate the symbolic searcher's functionality, and Sections 6 and 7 discuss conclusions and next steps after the hackathon.

2 Background

2.1 MEV quantification

MEV has generally been split into several well-known and commonly encountered categories–arbitrage, swap, liquidation, NFT–as well as a miscellaneous category of long-tail MEV that occurs infrequently and not based on any consistent pattern. Quantifying realized MEV has been an important research goal (a lower bound in 2021 of total realized MEV was $1 billion on Ethereum alone), and existing methods (e.g. mev-inspect-py) are capable of identifying and quantifying well-known MEV categories. However, they are not as well-suited for identifying and quantifying long-tail MEV, so estimates for realized long-tail MEV remain difficult to construct, and our understanding of long-tail MEV is sparse at best.

2.2 Symbolic execution and `hevm`

Symbolic execution is a method for executing all possible logic paths of a program. Unlike typical execution, which assigns "concrete" values to function arguments and as a result executes along a single logic path of the program at a time, symbolic execution assigns "symbolic", or variable, values

Encode x Wintermute MEV Hackathon 2022.

to function arguments so it can identify branches in the logic path and explore them concurrently and comprehensively.

Symbolic execution has traditionally been used as a tool for identifying bugs in programs that are difficult to write tests for. It has been applied by tools like Manticore by Trail of Bits to do the same with Ethereum smart contracts. To identify bugs, a symbolic executor will explore all the logic paths until it finds one which results in an unintended state (e.g. an exception or assertion violation), at which point it records the function arguments necessary for reaching that bug. One such tool for running symbolic execution on EVM bytecode is `hevm symbolic`, which we will use extensively.

Symbolic execution has also been mentioned by some as a technique that can be used to extract MEV, and it was recently used in the Flashbots Clockwork Finance Framework to identify certain well-known forms of MEV (e.g. arbitrage) by testing "templates" of MEV txns on a vulnerable smart contract. However, there are currently no known examples of symbolic execution being used to identify long-tail MEV, namely MEV that does not follow a template.

2.3 "Ethereum is a Dark Forest"

The article "Ethereum is a Dark Forest" presents a canonical example of long-tail MEV, the timeline (delinated by block height) and relevant details of which we summarize here.

Pre-`10741205` Someone sent $13,000 worth of LP tokens to the LGO/WETH Uniswap V2 token pair contract. Based on the logic of the contract, specifically the `burn()` function, anyone can withdraw that $13,000 worth of LP tokens. Dan Robinson et al. recognize the problem and devise a plan.

`10741205` Dan et al. create Burner Wallet #1 and transfer ETH to it.

`10741271` Dan et al. deploy a Setter contract using Burner Wallet #1.

`10741291` Dan et al. create Burner Wallet #2 and transfer ETH to it.

```
contract Getter is IGetter {
  IPool private pool;
  address private setter;
  address private getter;
  address private dest;
  bool private on;

  constructor(address pool_, address setter_, address getter_, address dest_)
     public {
    pool = IPool(pool_);
    setter = setter_;
    getter = getter_;
    dest = dest_;
  }

  function set(bool on_) public override {
    require(msg.sender == setter, "no-setter");
    on = on_;
  }

  function get() public {
    require(msg.sender == getter "no-getter");
    require(on == true, "no-break");
    pool.burn(dest);
  }
}
```

Listing 1: Snippet of the `Getter.sol` contract deployed in "Ethereum is a Dark Forest."

`10741326` Dan et al. deploy a Getter contract using Burner Wallet #2. The relevant code is included in Listing 1, as it will be referenced later in the paper. During deployment, the constructor sets `pool` to be the LGO/WETH token pair contract, `setter` to be the address of the previously deployed Setter contract, `getter` to be the address of Burner Wallet #2, and `dest` to be a new Burner Wallet #3. Once the `on` boolean is set to true by the Setter contract, the `get()` function can be called to rescue the $13,000 to the `dest` address.

`10741344` Dan et al. call the Setter contract to set the Getter contract to true. However, they fail to include the subsequent `get()` call in the same block, so a few blocks later they call the Setter contract again to reset the Getter contract to false and start over. This repeats two more times.

`10741410` Dan et al. call the Setter contract to set the Getter contract to true for the final time.

`10741413` Dan et al. send their `get()` txn using Burner Wallet #2 to the mempool. A generalized frontrunner sees the pending `get()` txn and calls the `burn()` function to withdraw the $13,000 first. Dan et al.'s `get()` txn reverts because the $13,000 is gone.

The MEV opportunity here requires a searcher to recognize that the `burn()` function can be called to produce risk-free profit. As an example of long-tail MEV, we will use it to demonstrate a symbolic execution approach to MEV detection. Unlike the existing generalized frontrunner, the symbolic searcher we construct can identify this MEV opportunity much earlier, before Dan et al. send their `get()` txn to the mempool at block `10741413`.

3 Symbolic execution for MEV

Theoretically, we can re-frame the MEV searching problem as a bug-finding problem, if we intentionally define the "bug" as a program executing in such a way that some EOA address gains risk-free profit from it (e.g. its WETH balance increases but none of its other token balances decrease). Given this definition, a symbolic executor will iterate through all the logic paths of a program and, upon finding such a "bug", will return to the user the function call and inputs required to produce the "bug", which is exactly the txn required to exploit the MEV opportunity.

3.1 A toy example

In the "Ethereum is a Dark Forest" example, we can inject the source code (Listing 2) with an assertion that fails if such a "bug" occurs. Namely, the injected code will raise an assertion violation if the WETH balance of the `dest` address has increased during the `get()` function call.

```
...
interface IWETH {
    function balanceOf(address owner) external view returns (uint);
}
...
function get() public {
    // WETH address
    address addr = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    uint256 orig_balance = IWETH(addr).balanceOf(dest);
    require(msg.sender == getter, "no-getter");
    require(on == true, "no-break");
    pool.burn(dest);
    uint256 new_balance = IWETH(addr).balanceOf(dest);
    assert(new_balance <= orig_balance);
}
...
```

Listing 2: Snippets of `Getter.sol` with injected code in **bold** checking if MEV is present.

Running the `hevm symbolic` command on this modified source code at the correct blockchain address and block height (e.g. block `10741412`), will produce an assertion violation. The assertion violation indicates that the symbolic executor has found the MEV opportunity mentioned in "Ethereum is a Dark Forest." If `hevm symbolic` is run at a block height when the `on` boolean is not set to `true` (e.g. block `10741409`), or at a block height after the MEV opportunity is gone (e.g. block `10741414`), no assertion violation will be found by the symbolic executor.

3.2 Generalizing from the example

The above toy example makes several assumptions which we must relax in order to generalize the symbolic execution method into a useful algorithm. We list each assumption and the intuition for relaxing it, and we formalize those intuitions into a proof-of-concept algorithm in Section 4.

1. **The only token balance checked is WETH, and the only EOA address checked is `dest`.** Without reading "Ethereum is a Dark Forest", it is not immediately recognizable that WETH is the right token and `dest` is the right address to check. To relax this assumption, we can try assertion statements that check all ERC-20 token balances (or some whitelist of ERC-20 tokens) for all addresses present in the smart contract's storage and the `msg.sender` address. Checking all these combinations is feasible because `hevm symbolic` runs off-chain.
2. **This does not check if an arbitrary EOA address that is not `dest` can exploit the MEV opportunity, or that the other token balances of that address do not decrease.** In fact, the whitehat engineers behind `Getter.sol` built it so that only the `getter` address is capable of calling the `get()` function and only the `dest` address is capable of receiving the funds. To relax this assumption, once an MEV opportunity is identified for some address and ERC-20 token through symbolic execution, we should concretely execute the same function call, but replace that address with an arbitrary address (e.g. our address) and check that (a) the ERC-20 token balance of the arbitrary address increases, and (b) none of the arbitrary address's other known token balances have decreased during the function call. If this concrete execution passes the checks, then an MEV opportunity exists.
3. **The source code of most smart contracts is not available to perform this code injection.** Only the run-time bytecode of a smart contract is stored on the blockchain. However, we can modify our injection such that it can be done on the contract bytecode *without ever viewing the contract source code*. Intuitively, we first shorten our Solidity code injection (see Listing 3) so that the subsequent bytecode injection can be cleaner.

```
...
interface IWETH {
    function balanceOf(address owner) external view returns (uint);
}
...
function get() public {
    require(msg.sender == getter, "no-getter");
    require(on == true, "no-break");
    pool.burn(dest);
    assert(IWETH(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2).balanceOf
    (0xe87e7073ec3021866553f0cdd73067a5aecc8e50) <= 0);
}
...

```

Listing 3: Snippets of `Getter.sol` with more concise injected code in **bold**.

Notice that this more concise injection hard-codes the values for `dest` and `orig_balance` to be `0xe87e7073ec3021866553f0cdd73067a5aecc8e50` and `0` respectively: more generally, we can always look up the address and its original balance of the relevant ERC-20 token beforehand and hard-code those values into our code injection. Second, we can isolate the now single line of injected Solidity code[1] which is of the form `assert(ERC20(addr).balanceOf(dest) <= orig_balance)`. We compile it into bytecode, and inject that bytecode into the run-time bytecode of any smart contract on the blockchain using a `bytecodeInjection` algorithm.

4 Prototype

We construct a proof-of-concept searcher that uses symbolic execution to identify long-tail MEV opportunities. It relaxes the assumptions outlined in the toy example above such that it can be applied

[1]The IWETH interface is necessary for compilation, but does not show up in the subsequent bytecode.

to identify MEV on any (possibly closed-source) smart contract, through any ERC-20 token, and for any EOA address. The pseudocode is presented in Algorithm 1.

Algorithm 1 A proof-of-concept symbolic searcher

```
Require: $t \geq 0$                                                    ▷ The block height $t$
 1: $C \leftarrow$ a contract on the blockchain
 2: $B \leftarrow$ bytecode($C$)                                ▷ The bytecode for $C$ is always available
 3: $A^* \leftarrow$ an arbitrary address (e.g. searcher address)
 4: for contract function $F \in C$ do
 5:     for ERC-20 token $T$ on the whitelist do
 6:         for address $A$ in {msg.sender, storage of $C$} do
 7:             $O \leftarrow T$.balanceOf($A$)                          ▷ The original token balance
 8:             $\hat{B} \leftarrow$ bytecodeInjection($B, T, A, O$)
 9:             Result, Inputs $\leftarrow$ hevmSymbolic($\hat{B}, F, t$)
10:             if Result is an assertion violation then
11:                 $F^* \leftarrow F$ with $A^*$ replacing $A$ and without all require function calls
12:                 $B^* \leftarrow B$ with $A^*$ replacing $A$ and without all require function calls
13:                 $O^* \leftarrow T$.balanceOf($A^*$)
14:                 $\hat{B}^* \leftarrow$ bytecodeInjection($B^*, T, A^*, O^*$)
15:                 Result$^*$, Inputs$^*$ $\leftarrow$ hevmSymbolic($\hat{B}^*, F^*, t$)
16:                 if Result$^*$ is an assertion violation and noTokenBalancesDecrease($A^*$) then
17:                     $N^* \leftarrow T$.balanceOf($A^*$) after $F^*$(Inputs$^*$)
18:                     MEV with volume $N^* - O^*$ available by deploying and calling $F^*$(Inputs$^*$)
19:                 end if
20:             end if
21:         end for
22:     end for
23: end for
```

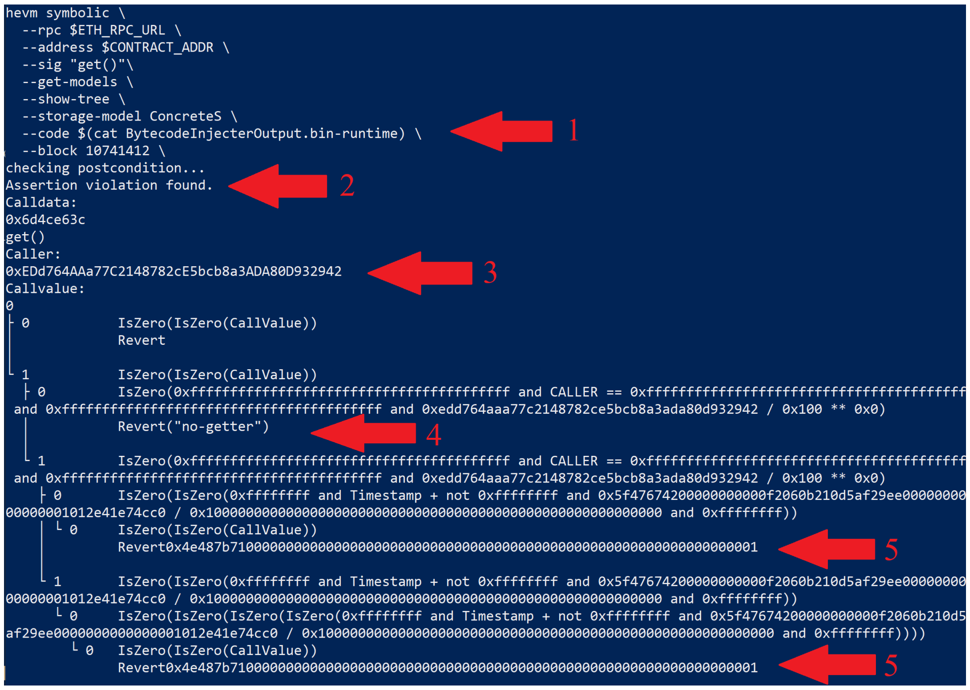
In Algorithm 1, the input is a smart contract C at a specific block height t, and an arbitrary address A^* we want to identify MEV opportunities for. Lines 4-6 iterate through the functions F of the smart contract C and all combinations of whitelisted ERC-20 tokens T and relevant addresses A, thus relaxing assumption (1). In lines 7-10, the algorithm uses `bytecodeInjection` and `hevmSymbolic` in conjunction with the bytecode B, not the source code, to check for MEV opportunities, thus relaxing assumption (3). If MEV is found, lines 11-19 replace the address A with our desired address A^* to check if the MEV still exists without decreasing other token balances of A^*, thus relaxing assumption (2). We remove all `require` function calls in F^*, B^* because they do not change the presence of MEV but would prevent address A^* from extracting it. If the MEV is confirmed, our final F^*(`Inputs`*) in line 18 of Algorithm 1 would likewise omit all `require` function calls.

Currently, an implementation of `bytecodeInjection` as well as a script for running lines 7-10 (the core symbolic execution technique) are available in our GitHub repository. After the hackathon, we expect to implement the rest of the pseudocode. The detailed logic behind `bytecodeInjection` can be found in Section 8.1 of the Appendix and in the implementation found on GitHub.

5 Application

We run the script demonstrating the core symbolic execution technique on the "Ethereum is a Dark Forest" MEV example. The `hevmSymbolic` command (line 15 of Algorithm 1) and subsequent output are shown in Figure 1. Inspecting the annotations, we make some observations:

1. The `--code` flag is followed by the contents of the run-time bytecode file `BytecodeInjectorOutput.bin-runtime`, which is where the bytecode output $\hat{B} \leftarrow$ `bytecodeInjection`(B, T, A, O) from line 14 of Algorithm 1 is saved after being algorithmically generated. Here, T, A, O are all as specified in Listing 3. Also note the `--block` flag followed by `10741412` to specify the correct block height for symbolic execution, as expected.



```
hevm symbolic \
  --rpc $ETH_RPC_URL \
  --address $CONTRACT_ADDR \
  --sig "get()"\
  --get-models \
  --show-tree \
  --storage-model ConcreteS \
  --code $(cat BytecodeInjecterOutput.bin-runtime) \
  --block 10741412 \
checking postcondition...
Assertion violation found.
Calldata:
0x6d4ce63c
get()
Caller:
0xEDd764AAa77C2148782cE5bcb8a3ADA80D932942
Callvalue:
0
├ 0          IsZero(IsZero(CallValue))
│            Revert
│
└ 1          IsZero(IsZero(CallValue))
  ├ 0        IsZero(0xffffffffffffffffffffffffffffffffffffffff and CALLER == 0xffffffffffffffffffffffffffffffffffffffff
 and 0xffffffffffffffffffffffffffffffffffffffff and 0xedd764aaa77c2148782ce5bcb8a3ada80d932942 / 0x100 ** 0x0)
  │          Revert("no-getter")
  │
  └ 1        IsZero(0xffffffffffffffffffffffffffffffffffffffff and CALLER == 0xffffffffffffffffffffffffffffffffffffffff
 and 0xffffffffffffffffffffffffffffffffffffffff and 0xedd764aaa77c2148782ce5bcb8a3ada80d932942 / 0x100 ** 0x0)
    ├ 0      IsZero(IsZero(0xffffffff and Timestamp + not 0xffffffff and 0x5f47674200000000000f2060b210d5af29ee00000000
0000001012e41e74cc0 / 0x100000000000000000000000000000000000000000000000000000000 and 0xffffffff))
    │ └ 0    IsZero(IsZero(CallValue))
    │        Revert0x4e487b710000000000000000000000000000000000000000000000000000000000000001
    │
    └ 1      IsZero(IsZero(0xffffffff and Timestamp + not 0xffffffff and 0x5f47674200000000000f2060b210d5af29ee00000000
0000001012e41e74cc0 / 0x100000000000000000000000000000000000000000000000000000000 and 0xffffffff))
      └ 0    IsZero(IsZero(IsZero(IsZero(0xffffffff and Timestamp + not 0xffffffff and 0x5f47674200000000000f2060b210d5
af29ee0000000000000001012e41e74cc0 / 0x100000000000000000000000000000000000000000000000000000000 and 0xffffffff))))
        └ 0  IsZero(IsZero(CallValue))
             Revert0x4e487b710000000000000000000000000000000000000000000000000000000000000001
```

Figure 1: The call to `hevm symbolic` which locates an assertion violation, with annotations in red.

2. The symbolic executor finds `Result` ← `assertion violation`, meaning the WETH balance of the `dest` address (Burner Wallet #3) increased during the `get()` function call.
3. The symbolic executor displays the txn inputs `Inputs` that resulted in this assertion violation. This includes the caller address: it correctly detects that the only caller address which results in the assertion violation (MEV) is `getter = 0xEDd76...` (Burner Wallet #2).
4. The symbolic executor then displays the full logic tree of the bytecode. In this case, each branch ends in a `REVERT` opcode. This particular branch corresponds to the logic path where the caller address is not `getter`, in which case the smart contract reverts with the error message "no-getter."
5. The symbolic executor displays two logic paths which both end in a `REVERT` opcode with Panic code `0x01`. This is our user-defined assertion violation where MEV was identified.

If the same `hevmSymbolic` command is run but at a greater block height (e.g. `10741414`), no assertion violation is found by the symbolic executor, because the generalized frontrunner eliminated the MEV opportunity at block height `10741413` already.

Unlike the generalized frontrunner, however, this symbolic searcher would have, if deployed at the time, identified the MEV opportunity as early as block height `10741344`, right after the `on` boolean was first set to true, instead of reacting only when the `get()` txn later reached the mempool right before block height `10741413`. Thus, our proof-of-concept symbolic searcher is able to algorithmically inject the relevant bytecode to run symbolic execution and subsequently identify this instance of long-tail MEV.

6 Conclusion

Symbolic execution can be used to identify and quantify certain types of long-tail MEV which existing methods cannot. During this hackathon, we built a theoretical framework for applying symbolic execution to MEV, implemented a proof-of-concept symbolic searcher, and successfully identified the canonical "Ethereum is a Dark Forest" MEV opportunity using it. As far as we know, this method of injecting bytecode for symbolic execution to identify long-tail MEV has not been publicly available prior to our research.

7 Implementation

There are two directions to fully implement the symbolic searcher after the hackathon.

The first direction is to quantify a lower bound on the total historical long-tail MEV present on Ethereum. This can be done by running the symbolic searcher on all, or a sampling of all, historical Ethereum txns. Such a research finding would greatly improve our understanding of historical MEV and its effects on Ethereum. It would allow us to build a useful dichotomy of long-tail MEV opportunities distinguishing those that can be detected by symbolic execution from those that cannot.

The second direction is to work on turning the symbolic searcher into a general-purpose MEV searcher with improvements to its MEV detection methodology. It has the potential to identify and exploit future MEV opportunities without relying on access to pending txns in a mempool, something generalized frontrunners require, and without relying on a well-known template for MEV, something arbitrage and liquidation bots require.

8 Appendix

8.1 Implementation of `bytecodeInjection`

We present the pseudocode for `bytecodeInjection` in Algorithm 2 along with the Solidity code created by `injectedAssertContract` in Listing 4.

Algorithm 2 for `bytecodeInjection`

```
 1: $B \leftarrow$ bytecode($C$)
 2: $T \leftarrow$ ERC-20 token $T$                              ▷ addr in code
 3: $A \leftarrow$ address                                      ▷ dest in code
 4: $O \leftarrow T$.balanceOf($A$)                             ▷ orig_balance in code
 5: $C_i \leftarrow$ injectedAssertContract($T, A, O$)          ▷ see Listing 4
 6: $B_i \leftarrow$ bytecode($C_i$)
 7: $\hat{B} \leftarrow B$
 8: for STOP opcode in $B$ do          ▷ Evaluate the assertion before every STOP opcode
 9:     $\hat{B} \leftarrow \hat{B}$ concatenated with a copy of $B_i$
10:     Redirect the most recent JUMP opcode index in $B$ (now in $\hat{B}$) to point to the copy of $B_i$
11:     Update all JUMP opcode indices in the copy of $B_i$ to account for being concatenated to $\hat{B}$
12: end for
13: Return $\hat{B}$
```

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.8.17;

interface IERC20 {
    function balanceOf(address owner) external view returns (uint);
}

contract InjectedAssert {
     function get() public {
        assert(IERC20(addr).balanceOf(dest) <= orig_balance);
     }
}
```

Listing 4: The `InjectedAssert.sol` file created by `injectedAssertContract` for a given hard-coded `addr, dest, orig_balance` specified in Algorithm 2.

The pseudocode first turns the desired assertion, with hardcoded token, destination, and original balance, into bytecode B_i. Then, it reroutes every `STOP` opcode in B to run a copy of the assertion bytecode before terminating. Each such copy is concatenated to a growing $\hat{B}$. Note that copying B_i many times is not expensive because we are only running this bytecode off-chain using `hevm`. Several implementation nuances are omitted here but documented in the GitHub repository.