6.857 Final Project

Exploits on Ethereum

Introduction

The Ethereum Virtual Machine (EVM) was launched in 2015, with the objective of implementing a generalized state machine, consisting of a globally-accessible singleton state mutated through the transactions of its accompanying virtual machine. These transactions, grouped into blocks, are punctuated by incentives for nodes to mine, necessitating a systemic way to transmit value. Ethereum addresses this requirement through its intrinsic currency: Ether.[1]

The Ethereum Virtual Machine (EVM) is quasi-Turing complete, and though it is therefore capable of a variety of functions, its most common uses involve financial instruments. As a consequence, security exploits and attacks on Ethereum are most typically directed at acquiring ownership of Ether by initiating a transfer from the target's wallet to the adversary's. This focus and the role of Ether in the underlying blockchain makes it possible to directly quantify value, providing the basis for an objective measure of scale for the exploitation of historical security failures.

This paper not only examines prominent or representative exploits of these attack patterns but also identifies the countermeasures developed to address the underlying vulnerabilities, and their present feasibility. This survey of historical Layer 1 Ethereum exploits precedes discussion of the increased complexity, and thus vulnerability, produced by Layer 2 systems, followed by prognostication on the future of Ethereum's security as a consequence of potential alterations the EVM may undergo in the near future.

Ethereum Security Policy

As mentioned previously, the EVM is quasi-Turing complete. This makes it difficult to enumerate the totality of the security policy, but the most important axioms are perhaps "code is law," and "access implies permission." Programs on Ethereum are written in Solidity, constructed as "smart contracts," which range from public libraries meant simply for invocation to wallet managers and other applications with their own intended flow and structure. Any public function within these contracts is capable of being invoked from anywhere, and functions are public by default.

Thus if a contract contains a function which transfers ether out of the contract's wallet and to another, all such transactions generated by executions of that function are legitimate within the

[1] http://cryptochainuni.com/wp-content/uploads/Ethereum-A-Secure-Decentralised-Generalised-Transaction-Ledger-Yellow-Paper.pdf

confines of the EVM. Regulation of the rate or destination of these transactions are solely the responsibility of the contract and its developer. Failure on the part of the developer to regulate the contract's transfers of ether is in no way impingement on the Ethereum security policy.

To employ the actor and permission model from Lecture One, the Ethereum security policy is that all withdrawals from any wallet can only be made with access to the owner's wallet.

Overview of Layer 1 Exploitations

We build upon the categorization created by Chen et al., which organizes the vulnerabilities of Ethereum into three broad causal categories: Ethereum design and implementation, Solidity language and toolchain, smart contract programming, and of course, human factors.[2] We employ more narrow categorizations based on similarities between attack vectors not enumerated by their survey, and examine a few such constructed enclaves.

The following sections are an inexhaustive list of types of attacks and exploits on Ethereum's application layer, accompanied by historical examples. Where helpful, code snippets or attack flows are provided to explain the vulnerability and its exploit.

Scheduling and Timing Attacks

The EVM is characterized by a multitude of users operating synchronously on the underlying blockchain. In order for the EVM to resolve to a singleton, or canonical state, it is necessary to produce a strict ordering of all the transactions. There are a multitude of potential problems relating to the consensus construction of the EVM's transaction scheduling. We enumerate three such problems amenable to exploitation, all of which are the consequence of how Ethereum handles transactions, a key feature of its design.

In order to handle transactions, Ethereum nodes collect transactions into blocks, only confirming those transactions once a miner has solved a consensus mechanism (presently this is proof-of-work). This system gives rise to a type of race condition commonly referred to as a Transaction Ordering Dependance. The miner in charge of that block of transactions chooses which transactions are included and how they are ordered, typically by the "gasPrice" associated with the transactions. The higher the gasPrice, the greater the incentive a miner has to include those transactions in the block; upon a block's mining, the miner affixes a block timestamp and receives the gasPrice of all the included transitions as payments in Ether.

Front Running

Any adversary can exploit this system through an attack known as Front Running. The adversary monitors the transaction pool for transactions of interest, such as those containing

---

[2] https://dl.acm.org/doi/fullHtml/10.1145/3391195

solutions to problems, or those that change the adversary's permissions or state in a contract in some manner undesirable to the attacker. Using the data from this observed transaction, the attacker then creates their own transaction with a higher 'gasPrice' to get their transaction included in a block before the original.

Example

As an example of such an attack, consider the ERC20 standard commonly used for building tokens. The approve() function allows a user to grant permission to another user to transfer tokens on their behalf.[3] Here is an attack that has since been addressed:

1. Alice approves Bob to spend 100 tokens.
2. Alice revokes Bob's approval to spend 100 tokens by creating a transaction that sets Bob's allocation to 50 tokens.
3. Bob has been watching the chain and sees this transaction. Bob builds a transaction of his own spending the 100 tokens, placing a higher gasPrice on his transaction than Alice's so his transaction goes through before hers.
4. Bob's transfer of 100 tokens goes through before Alice's transaction gets committed.
5. Alice's transaction sets Bob's allocation to 50 tokens, thereby giving Bob 150 tokens inadvertently.

Present Feasibility

A commit-reveal strategy can help prevent Front Running exploits. Users send transactions with hidden information (a hash). After the transaction has been included in a block, the user sends a transaction revealing the data that was sent. This method prevents miners and users from frontrunning transactions as they cannot determine the contents of the transaction. This method however, cannot conceal the transaction value, which in some cases is the valuable information that needs to be hidden.

This shortcoming can be worked around, as exemplified through the ENS smart contract, which lets users send transactions of hidden data that includes the amount of ether they are committing to spend[4]. Users then send transactions of arbitrary value and are refunded the difference between the amount sent in the transaction and the amount they committed to spend once that data is revealed.

---

[3] https://docs.openzeppelin.com/contracts/4.x/erc20
[4] https://ens.domains/

Timestamp Manipulation

Adversaries who are operating mining nodes have two more, closely related attacks produced by their control over block creation; the timestamp. Block timestamps can be manipulated by miners subject to two constraints. First, that the fraudulent time stamp does not preempt the timestamp of the parent block, and secondly, that the manipulated timestamp does not too drastically precede the true time.

Attacks

This ability to alter timestamps means that programs that use time-based logic can be easily exploited by miners, targeting the timestamp's two most common uses in contracts: as the input to time-based conditionals or as a source of entropy. A common conceptual example of the latter attack is a roulette program that determines the final position of the ball based on a pseudorandom value seeded by the block's timestamp. A miner can alter the timestamp of the submitted block to make the contract hit their bet, and will subsequently be rewarded with ether.

Example

A real world example of the former style of timestamp manipulation attack was a game called GovernMental. Users would "join" during a round, and the last player to join during a round would be rewarded Ether. Joining times were determined by block timestamps, which enabled players to make it seem like they were the last player to join during a round, cheating the game and accumulating rewarded ether unfairly.

Present Feasibility

Sources of entropy remain an issue for the EVM. As a consequence of EVM's determinism and lack of ability to store any data secretly, potential sources of entropy must be independent of prior state, preventing the use of a pseudorandom function to generate a series of random numbers.

Solidity Honeypots

As with any coding language, the nuances of Soldity's behavior can be a source of issues for developers unfamiliar with the language. Some of these issues have been solved, while some remain pitfalls for the unaware. Of course, what can be done accidentally can also be done with intent, and these vulnerabilities are often exploited by the intentional construction of honeypots.

Uninitialized Storage Pointers

This exploit relies on the behavior of Solidity's data management mechanisms. At a high level, global memory available to all functions within a contract is held in storage, which constitutes a

series of 256-bit slots filled by variables in the order of their appearance in the contract. Uninitialized local storage variables point to storage, and by proxy, to slot 0. This results in unintended aliasing.

This aliasing allows for the potential manipulation of the contract's state in manners unexpected by the developer, or enables malicious developers to exploit this vulnerability at a later time. This vulnerability lends itself to a honeypot attack vector, as seemingly harmless functions which manipulate an uninitialized struct with N fields will actually be manipulating the data in the first N slots of storage, potentially causing unexpected behavior or accidentally violating invariants the contract relies on. It is easy to imagine how control over some of the data in storage may be intended to be locked behind some authentication, but with an uninitialized struct, these authentications can be bypassed.

Example

OpenAddressLottery and CryptoRoulette, both honeypots, employed this attack vector to drain the wallet.

Present Feasibility

As this exploit is a consequence of aliasing produced by Solidity's default behavior, the best practice is for the developer to simply avoid declaring uninitialized storage variables. In order to encourage this, the Solidity compiler began raising uninitialized storage variables as errors, starting with version 0.5.0.[5] However it must be noted that adversarial developers can continue to employ older versions of Solidity or suppress these errors.

Type Casting

Many contracts in Ethereum utilize the ability to call functions defined in other contracts. One way this is accomplished is by passing the address of the callee contract into the constructor of the caller contract. A toy example of such an arrangement is demonstrated in the code block that follows:

[5] https://docs.soliditylang.org/en/v0.5.0/050-breaking-changes.html

```
contract A{
      function transferOneEther(){
        //transfers one Ether to accounts
      }
}
contract B{
      A a;
      constructor(A _a){
         a = _a;
      }
      function doNTransfers (uint n) returns (uint){
            //calls a.transferOneEther() n times
      }
}
```

In a proper implementation of contract B, when doNTransfers is called, exactly N ether will be transferred using contract A's transferOneEther function, One pitfall with this approach, however, is that Solidity features automatic type-casting, allowing contracts that are not of type A to act as contracts that are of type A. An adversary can create a contract like the contract E below, pass it as the input to B's constructor, and thereby magnify the returned Ether by 100.

```
contract E{
      function transferOneEther(){
            //transfers 100 ether
       }
}
```

Present Feasibility

This toy example is a very simple case in which detection of the vulnerability is straightforward, however in more complex cases, this acts as a way for contract owners to hide malicious code by masking operations under different names, producing a honeypot that may be exploited later.

The key preventative measure to prevent the exploitation of this vulnerability is to ensure that instances of contracts are not passed into the constructors of other contracts when unnecessary, and instead are manually constructed within the contract. This is largely a doctrinal solution, and cannot be programmatically enforced. As such, this remains an attack vector ripe for future exploitation.

Smart Contract Publicity

As mentioned previously, functions within smart contracts on Ethereum are public by default, a design principle promoting the reuse of code across EVM. What this means for the security of Ethereum is that without active visibility management, smart contracts present a broad attack surface, and developers of smart contracts must be careful to consider the ramifications of invocation flows through their contract beyond those envisioned for its correct usage.

DAO Hack

One of the most famous hacks to ever happen in the Ethereum ecosystem, the DAO hack resulted in the theft of 3.64 million ether by a single hacker, a sum then constituting 15 percent of all circulating ether in early 2016[6]. This attack was so devastating that it resulted in the “roll back” of Ethereum – the creation of a new system consensus in which the stolen ether was removed from the wallet of the attacker and restored to the original holders.

This hack was achieved by exploiting the public nature of contract functions, in a style of attack known as a re-entrancy attack, so called because it exploits the re-entrancy vulnerability of smart contracts’ public functions.

Re-Entrancy Vulnerability

The re-entrancy attack was as follows: The DAO had a function to allow users to withdraw money they had earlier put into the system, a simple mock-up of which is below:

```
1  contract BasicDAO {
2
3    mapping (address => uint) public balances;
4    ...
5
6    // transfer the entire balance of the caller of this function
7      // to the caller
8    function withdrawBalance() public {
9      bool result = msg.sender.call.value(balances[msg.sender])();
10     if (!result) {
11       throw;
12     }
13   // update balance of the withdrawer.
14     balances[msg.sender] = 0;
15   }
16 }
```

In line 9, this code sends the user whatever money they had deposited. Line 10 only checks to see if the transfer of ether worked. Then in line 14, the user’s new balance is updated to 0 so they can’t double spend.

---

[6] https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/

The attacker took advantage of the ordering of these statements. He wrote a proxy contract to withdraw money instead of initiating the withdrawal with a regular account, so that he could have the proxy contract trigger a function of its own upon receiving the transferred ether.

First, the attacker's contract deposits money into the DAO. Then it fairly asks to withdraw the money. Line 9 is executed in the DAO contract, which, in turn, triggers the attacker's proxy contract; before the DAO code can progress to line 14, the hacker's smart contract is once again asking to withdraw his balance, the same money he's already received.

This loop continues until there have been enough iterations for the hacker to receive millions of ether. Once it terminates, the DAO code finally progresses to line 14 and updates the attacker's balance to 0, too late to prevent the double spending that instantly made the hacker a millionaire.

Prevention

After a recommendation from Vitalik (one of the founders of Ethereum) to do so, the Ethereum community voted with their money and with their mining resources, and a new blockchain was created where all the money hacked was taken from the hacker's wallet and put into a new safe contract where the money could be fairly withdrawn by the participants.

The old blockchain, however, still had supporters who believed it was the one "real" blockchain, and to this day exists as Ethereum Classic, the "original" blockchain.

This is an interesting example of how public consensus can "fix" almost any blockchain bug if enough people agree that something unfair has happened and they want to ignore the result of code even if it obeyed the fundamental rules of the EVM.

On the other hand, it shows that given enough incentive, miners and Ethereum validators can choose to change any number of transactions to whatever state they wish, as long as they control enough of the popular vote.

Layer 2 Security

As the number of people using Ethereum increases, so does the demand for data to be processed. As a result gas prices have grown and transactions have become significantly slower. These inefficiencies introduce the need for data scalability solutions.

One of the most widespread solutions implemented today is known as Layer 2 scaling. Layer 2 scaling is a term for any solutions that process transactions off chain before finally adding them to the Ethereum Blockchain. Currently, the main blockchain can handle transactions at a rate of approximately 15 transactions per second.[7] By using Layer 2 scaling solutions, this number is estimated to increase to 2,000-4,000 transactions per second, a substantial boost in efficiency.[8]

---

[7] https://ycharts.com/indicators/ethereum_transactions_per_day
[8] https://finematics.com/ethereum-layer-2-scaling-explained/#:~:text=Layer%202%20scaling%20can%20

Layer 2 solutions are best thought of as sitting atop Layer 1 (the main blockchain). Layer 2 solutions do not require changes to Layer 1, but simply pass packaged data into existing Layer 1 mechanisms. This property allows for a large number of Layer 2 solutions to exist, explaining the increase in magnitude of the transaction throughput. Additionally, since packaged data is being submitted to Layer 1, Layer 2 solutions leverage the security of Layer 1, from which they derive security.

Although Layer 2 solutions can take on various different forms, they generally follow a basic blueprint centered around a group of server nodes. Instead of transactions being submitted directly to the main blockchain, transactions are instead sent to these server nodes, where they are then batched into groups and sent to be processed on the main chain.

This batching process creates a structure known as a rollup, which acts as a summary of the transactions processed in a Layer 2 solution. However, a natural question arises from the trustlessness of the blockchain: how does the main blockchain know whether a rollup contains information about real transactions, or if the rollup was sent by a malicious actor? This is where it becomes important to differentiate the different types of rollups; as of now, there are two main rollup types: optimistic rollups and Zero Knowledge rollups (ZK-rollups).

Rollup Types

Optimistic rollups post data to Layer 1 under the assumption that the data is correct (hence the name "optimistic"). If the data is correct and nobody believes it is fraudulent, then no additional work is done and the rollup transaction remains in Layer 1. If instead, the optimistic rollup is fraudulent, another party can submit a fraud proof. If a fraud proof is verified, this necessarily means that the rollup contains fraudulent data, and the issuing party is financially penalized through the loss of their fidelity bond. This process of dispute resolution is computationally expensive, and as a result should not be performed very often. Optimistic rollups are effective when parties submitting rollups are mostly trustworthy, since little extra work has to be done in the case data is correct. However, if there are many bad actors in this system, many iterations of dispute resolution will have to be performed resulting in the use of a lot of the main chain's computational power, decreasing overall transaction throughput and speed, which were the two factors that this scalability solution seeks to improve.

The alternative type of rollup, known as a ZK-rollup, utilizes cryptographic proofs in order to circumvent this problem. When a Layer 2 solution posts a ZK-rollup onto the main chain, it includes a cryptographic proof called a ZK-SNARK in the batch data. ZK-SNARKs can be quickly verified by Layer 1 nodes in order to identify validity. The drawback of using ZK-rollups is that they require a tremendous amount of precomputation in order to generate a ZK-SNARK. The difficulty of producing ZK-SNARKS incentivizes Layer 2 solutions to employ optimistic rollups, which consequently employed more often within Layer 2 solutions.

---

dramatically,2000%2D4000%20tx%2Fsecond.

Arbitrum

Arbitrum is one of the most popular options when it comes to Layer 2 solutions using optimistic rollups. After collecting a group of transactions and creating an optimistic rollup for them, Arbitrum places these optimistic rollups on a proprietary side chain. All dispute resolutions happen on this side chain. This promotes the efficiency of the process, as the main chain is only affected if the rollup is not found to be fraudulent.

But on what basis does the determinations of the side chain gain validity? Arbitrum employs certain nodes as validators, which are selected through stakes of Ether. If all validators agree that all transactions within a block are valid, the transactions are given a stamp of having the "AnyTrust Guarantee."

Many decentralized finance protocols integrate Arbitrum, most notable of which is SushiSwap. According to DeFi Llama, an aggregator of decentralized finance data, around $1.69 Billion are locked within Arbitrum contracts, of which SushiSwap accounts for 24%.[9] Arbitrum contracts have the potential to hold huge amounts of money, which is why it is important that solutions built on top of Arbitrum to have airtight security.

Case study: TreasureDao Hack 2022

Earlier this year, the biggest NFT marketplace on Arbitrum known as TreasureDao faced a massive security breach. More than 100 popular NFT's were stolen from the marketplace, 17 of which valued on the order of several thousand dollars. The total value of the NFTs came out to be $1.4 million, which was a major loss for the TreasureDao marketplace. The protocol's token, called MAGIC, suffered a massive loss in value exceeding 33%, reducing the value of each token being worth $3.82 to a new value of $2.55.

This hack was enabled by TreasureDao's use of an interface called ERC-721, which required TreasuerDao's smart contract to specify how trading of NFTs worked. Through a bug in the buyItem() function, the hackers were able to trick the protocol into thinking it implemented an interface called ERC-1155 instead. This interface is very similar to ERC-721, except that it specifies behavior for all different types of tokens, not just NFTs. Since the protocol did not actually implement ERC-1155, the hackers were able to pass in whatever values they wanted into this buyItem() function, resulting in them being able to "buy" the NFTs for free.

This hack is an example of how layer 2 solutions can present extra security issues not present in Layer 1. As one of the common sayings in cryptography goes, complexity is the enemy of security. By building a framework on top of the preexisting Layer 1 functionality, Layer 2 contains the previously enumerated exploits achievable through Layer 1 in addition to brand new exploits introduced in Layer 2. Layer 2 inherits Layer 1's security, and its insecurities too.

---

[9] https://defillama.com/chain/Arbitrum

Polygon (MATIC)

Polygon, formerly known as MATIC, is another popular Layer 2 solution, presently possessing the highest market share out of all existing Layer 2 solutions. Whereas Arbitrum is a singular scaling solution, Polygon's goal is to be an ecosystem of different scaling solutions. Polygon promotes this model by providing a framework empowering users to create their own scaling solutions.

The ecosystem architecture is split up into four layers, each of which has different functionalities and different security guarantees. The first layer, known as the Ethereum layer, leverages the use of smart contracts directly on the main block chain. This is one of the most straightforward scaling solutions, where transactions are processed within this layer and directly added onto Layer 1 once all conflict resolution stages are complete. Since this layer adds the least complexity on top of the Layer 1 Ethereum structures, the Ethereum layer of Polygon most closely retains the security of the main Ethereum blockchain. This layer is not mandatory within solutions that build on the polygon framework, and is generally used in protocols that value security and store money in smart contracts.

The security layer is another non mandatory layer that provides validation as a service function. Scaling solutions that require validators in order to operate (such as with the rollups explained previously) can use this service offered by Polygon rather than building their own validator service. An example of a protocol that could utilize this layer would be an NFT marketplace, which necessitates that transactions are properly validated. The security layer can take on any number of different implementations, and can even use miners on the main chain as validators in some implementations.

The third layer is known as the Polygon Network Layer. This is a mandatory layer within Polygon scaling solutions, as it is where transactions are processed. Transaction processing is divided into three steps: transaction collation, which is the collection of transactions from external parties; block production, which is the packaging of these transactions into blocks; and consensus, which makes sure all nodes within this polygon scaling solution reach consensus about the data stored.

The final layer, also mandatory, is known as the execution layer. This layer is responsible for executing the transactions collected and packaged within the Polygon Network Layer.

One observation about the Polygon architecture is that it is very abstract. This is deliberate as Polygon is trying to create a generalized framework for applications to implement scaling solutions that suit their needs. Since different applications look to optimize different features, such as security, speed, cost, or sovereignty, Polygon provides methods to maximize the features that the application values the most.

Another observation would be that this approach massively expands the attack surface, while also relegating much of the responsibility for the security of any resultant systems to the developers of those systems.

MEV

A great example of the exploits further enabled by Layer 2 is Miner Extractable Value, a category of profitable transaction reorderings that can be made by miners in order to extract more value from transactions than the typical mined rewards. To make this more concrete, we will analyze MEV use-cases in transaction reordering and front-running on decentralized exchange protocols on Ethereum.

Decentralized exchanges on Ethereum, such as Uniswap and Sushiswap, are used in order to make anonymous and permissionless swaps from any token *x* to any token *y* that has liquidity provided for it by other users. These providers put down liquid capital that to be used for the market-making of the various trades on the decentralized exchange, or DEX's.

On DEX's that use a constant product formula, liquidity providers put in tokens of two different coins in equal-value amounts, and they're incentivized to do so by fees paid by those using the liquidity to swap in and out of token pairs.

When new liquidity is input into the system, the total liquidity *k* of the pool increases. If they later want to remove their tokens, they can take them out, and the total liquidity *k* of the pool decreases, although only proportional to their "fair share" of the market, as they can't withdraw more liquidity (and thus valuable tokens) than they originally input.

$$x \times y = k$$

TOKEN X QUANTITY
TOKEN Y QUANTITY
CONSTANT

This is the basic setup of a constant product market-maker, a simple mathematical implementation of an exchange originally proposed by Vitalik on a Reddit post.[10]

The "constant" *k can* change when liquidity-providers input value or remove their tokens again, but liquidity providers are only a small part of the ecosystem. The majority of the transactions are swaps done by users of the system who are paying the liquidity provider fees in order to use their liquidity. Whenever those users swap back and forth between tokens, the value *k* must be the same at the

[10] reddit.com/r/ethereum/comments/55m04x/

end of the transaction as it was at the beginning.[11] This property is why “constant” is used to describe *k*.

To further make our explanation concrete, we shall say that Alice is wanting to buy more Bitcoin (token *x)*, with Ether (token *y)*. You can trade “wrapped” versions of Bitcoin on these exchanges, making this a very common trade on DEX’s.

Looking at our constant product formula, whenever Alice wants to buy more Bitcoin with their Ether, they must make sure that at the end of the day, the amount of Bitcoin remaining in the pool multiplied by the amount of Ether remaining in the pool stays a constant *k*. If they try to take an amount of Bitcoin while giving an amount of Ethereum that doesn’t preserve the constant *k* by the end of the transaction, the entire transaction is rejected by EVM.

Observe that if Alice uses her Ether to buy more Bitcoin, there will be a consequential increase in the ‘price’ of Bitcoin, which we can model as the amount of Ether the next user will need to exchange for a Bitcoin. Conversely, the price of Ether decreases. The greater the amount of ether Alice spends to purchase Bitcoin, the higher the subsequent increase in Bitcoin ‘s price.

Looking at the formula, we can tell that the exact increase in price doesn’t depend only on how much Alice buys, but it also depends heavily on the constant *k*. The greater the value of k is, the greater the size of the liquidity pool provided for swapping is.

If there’s a large amount of liquidity in the pool, it generally indicates that many people are trading between these two assets, as liquidity providers flock to them for the increased traffic and thus increased fees being paid out. As an example, the Bitcoin and Ethereum liquidity pool will likely be one of the largest liquidity pools available as opposed to that of obscure ERC-20 tokens.

Alice's swap to bitcoin will not make as large of a splash in this giant pool and so while it won't push the price up as much, it will impact the price of the user’s desired coin at least slightly, and as there are multiple different DEX’s on Ethereum, this creates arbitrage opportunities when the price on different exchanges is mismatched.

Other actors on Ethereum can exploit this difference in price by buying the asset at the cheaper price and then selling it at the higher price until they once again balance price points across the various exchanges. So essentially, as soon as Alice makes a large purchase of Bitcoin, a race begins to balance the prices across all of the exchanges again. Whoever is first to balance gets the entire net-difference profit, and with cleverly written code that checks to make sure a profit is made before completing the transaction, the only risk taken by arbitrage exploiters is that of paying the miners for the gas fees of a failed transaction. Since failed transactions can still be included in the blockchain (although essentially with no effect besides costing the caller transaction fees) they still cost

---

[11] This is not actually true. It’s a slight simplification of the actual model, in which k must increase very slightly (in Uniswap and Sushiswap this value is 0.30%, such that the liquidity providers are given a reward for placing down their capital for the use of others.) In fact, k could increase as much as the swapper wanted it to, 0.3% is the minimum, but if users were feeling charitable towards providers, or more likely, if they made an error, they could increase k as much as they wanted as there’s only a check to make sure that k NEVER decreases through swapping and it increases by at least 0.3% of the swapped value each time.

computing power and thus the failed transactions spend some (usually not all) of the gas fees provided.

This puts miners in an extremely favorable condition, as they control the ordering of transactions, and can thus ensure they're the first to exploit any arbitrage opportunity taking place from large purchases of tokens. They also don't have to worry about paying the failed transaction fees.

Furthermore, even if miners don't discover this arbitrage opportunity themselves by monitoring various public APIs, when another user submits a transaction taking advantage of an arbitrage opportunity that they've spotted, they're essentially pointing out to the miners the location of free money.[12] The miners don't even have to find these opportunities themselves, they can simply copy the code that the user submits to be included in a block, and instead execute it themselves with the return address being their own wallet. (This is the front running exploit identified previously in Layer 1.)

Copying the profitable moves of others on the blockchain, purposely pushing up the price of assets in order to make profits from incoming large purchases, and immediately running the arbitrage upon someone else buying an asset (a practice known as back running), are all some of the major ways that miners and those colluding with or bribing miners can extract value from the Ethereum ecosystem.

All together, MEV has net over half a billion of profit at the time of writing this paper,[13] and this is only the MEV that researchers have been able to track and categorize, which leaves out a huge swath of possibly unnoticed exploitation. This is a vastly under-explored territory, and researchers assess that they're currently tracking only a lower bound on total possible MEV.[14]

Any time a user executes any transaction on Ethereum, there is potential for miner-extractable value, and finding the best ways to reorder transactions and call your own transactions (of which there are essentially infinite possibilities on the quasi-Turing-complete[15] EVM), means that it's computationally infeasible to determine the total potential MEV. We do know that untapped MEV is a vast landscape, and as we progress in our understanding of Ethereum exploits, we must keep in mind the opportunistic front-runners lurking below the surface.

---

[12] https://medium.com/@danrobinson/ethereum-is-a-dark-forest-ecc5f0505dff

[13] https://explore.flashbots.net/

[14] https://research.paradigm.xyz/MEV

[15] The EVM is Turing complete, but we say "quasi" and "essentially infinite" because there's somewhat of a practical bound on the complexity of transactions given that more complex bytecode is vastly more expensive.

Ethereum 2.0: Proof-of-Stake, Casper FFG, LMD-GHOST

We discuss the adaptations we deem likely in Ethereum's near future and explore the exploits these modifications will be susceptible to.

Proof-of-Stake (PoS)

Proof-of-stake is a consensus mechanism used by some blockchain networks to achieve distributed consensus. It requires users to stake their Eth to become a validator in the network. Validators perform the same role as a miner in a proof-of-work blockchain, as they also order transactions and create new blocks so that all nodes can agree on the state of the network.

Validators are chosen at random to create blocks and are responsible for checking and confirming blocks they don't create. A user's stake is used as a way to incentivize good behavior through reward and to punish bad behavior through loss of stake. For example, a user can lose a portion of their stake for things like going offline (failing to validate) or their entire stake for deliberate collusion.

Unlike proof-of-work, validators don't need to use significant amounts of computational power because they're selected at random and therefore aren't competing. Additionally, validators don't need to mine blocks; they just need to create blocks when chosen and validate proposed blocks when they're not. This validation is known as attesting. Validators get rewards for proposing new blocks and for attesting to ones they've seen.

Casper the Friendly Finality Gadget (Casper FFG):

This transition to a new PoS-based blockchain will be driven by the Casper FFG system, granting the responsibility for the verification of new blocks of transactions to validators, with voting power proportional to the amount of Eth they put at stake. For example, someone who has deposited 64 ETH will have double the voting weight of someone who deposited the minimum staking amount of 32 ETH. Random committees of validators will be selected to propose new blocks and receive block rewards, which consist of transaction fees, for doing so.

In distributed networks, a transaction has "finality" when it's part of a block that can't change. To do this in proof-of-stake, Casper, a finality protocol, gets validators to agree on the state of a block at certain checkpoints. So long as 2/3 of the validators agree, the block is finalized. Validators lose their entire stake if they later try to revert this via a 51% attack.

Casper will also act as an intermediary step in making Ethereum become environmentally friendly. Casper's role as a selector will promote the blockchain's security. In acting as a bookkeeper of the Ethereum 2.0 ledger, Casper enables the quick removal and punishment of malicious validators. The penalty for cheating the rules is the validator's stake (in Eth), making

network transgressions egregiously expensive. Lastly, Casper will give Ethereum greater levels of decentralization. Presently, there is an imbalance of power on the network for those who have the resources to run pooled mining operations. In the future, anyone who can buy the minimum stake will be able to help secure the blockchain.

LMD-GHOST:

The Greedy Heaviest Observed Subtree (GHOST) protocol is a fork-choice rule algorithm. GHOST selects the head of the chain by choosing the fork which has the most votes, aggregated across all of that block's subtree (i.e. more of the latest messages support either that block or one of its descendants). The algorithm repeats this process until it reaches a block with no children, arriving at a singleton state.

Ethereum 2.0 employs a variation of GHOST called Latest Message Driven GHOST (LMD-GHOST). The idea is that when calculating the head of the chain, we only consider the latest vote made by each validator, and not any of the votes made in the past. This decreases the computation since the number of forks that need to be considered to execute the fork choice cannot be greater than the number of validators.

Consensus within Etherium 2.0 relies on LMD-GHOST to add new blocks and decide what the head of the chain is, and Casper FFG to make the final decision on which blocks are and are not a part of the chain. GHOST's liveness properties allow new blocks to quickly and efficiently be added to the chain, while FFG follows behind to provide safety by finalizing epochs.

Attacks on Etherium 2.0

Unlike the previous section on attacks, those considered in this section are theoretical, and as of yet have no historical examples. As such, we focus on their mechanical presentations.

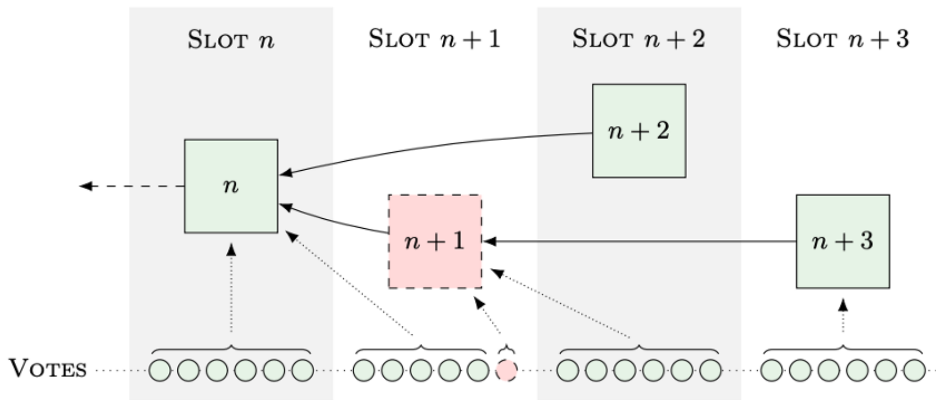
Reorganization (Reorg) Attack

The Reorganization Attack flow takes place over three slots, employing the attack flow enumerated below:

1. At the beginning of slot n + 1, the adversary privately creates block n + 1 on block n and privately attests to it. Honest validators do not see block n + 1 and so they attest to the previous head of the chain, block n.
2. At the beginning of the next slot, an honest validator proposes block n + 2. Assuming zero network latency, the adversary finally publishes their private block and attestation from slot n + 1 at the same time as block n + 2 is released. Honest validators now see both block n + 1 (and its one attestation) as well as block n + 2. These blocks are conflicting because they share the same parent, block n. Another result of sharing the

same parent is that block n + 1 inherits all the weight of block n, in particular the honest attestations from slot n + 1 voting for block n also count in favor of it. Hence, in slot n + 2 all honest validators vote for block n + 1 as head of the chain, because it has more weight due to the single adversarial attestation from slot n + 1.

3. Finally, at the beginning of slot n + 3, an honest validator proposes block n + 3 pointing to block n + 1 as its parent. This effectively orphans block n + 2 and brings the reorg attack to its conclusion.



The above strategy shows that a block proposer which controls a single committee member of the same slot can successfully perform a 1-reorg. This strategy can be extended to reorg attacks of arbitrary length k. Let the number of honest validators in any given committee be $W_{honest} \approx (1 - \beta)W \leq W$. Then, for a successful reorg attack of length k > 1, the proposing adversary needs to control $W_{honest}(k-1)+1$ validators, since it offsets honest committee members' votes in the first (k − 1) slots and uses the above refined attack strategy in the last slot.

Attack 2: Balancing Attack

1. The adversary waits for an opportune epoch to launch the attack. An opportune epoch is characterized by having adversarial block proposers in slot 0 and 1. Due to the random committee selection in PoS Ethereum, this happens with probability β^2 for any epoch, where β is the proportion of adversarial validators, so that the adversary needs to wait on average (1/β)^2 epochs until it can launch the attack.

2. Now assume epoch 0 is opportune. The adversarial proposers of slots 0 and 1 propose two competing chains – call them A and B. Note that this is not a slashable protocol violation. Both withhold their proposals so that none of slot 0 or 1 honest validators vote for either block.

3. The adversary releases the blocks after slot 1. Assume without loss of generality that when viewing A and B with the same number of votes, the protocol's tie-break chooses A. Then, a handful of adversarial votes per slot, released under carefully chosen circumstances, suffice to steer honest validators' votes so as to keep the system in a tie between the two chains and consequently stall consensus. Time T_{delay} before honest validators in slot 2 vote, the adversary releases a vote for B from an adversarial committee member of slot 1 (called a sway vote). If T_{delay} is tuned to the network propagation behavior, then roughly 1/2 of honest committee members of slot 2 see the sway vote before they cast their vote, and thus view B as leading (due to the sway vote) and will vote for it; and the other half see the sway vote only after they cast their vote, and thus view A as leading (due to the tie-break) at the time of voting and will vote for it.
4. Modeling the above as voting according to a coin flip for each validator, $\sim W_{honest} / 2$ of W_{honest} honest validators per slot would vote either A or B, with a gap of $O(\sqrt{W_{honest}})$ from the variance of a binomially distributed random variable. So, $O(1/\sqrt{W_{honest}})$ adversarial fraction of stake would work to rebalance the vote to a tie and keep the system in limbo. The adversary can reliably determine the time T_{delay} it takes for approximately 1/2 of nodes to receive a message broadcast by the adversary.
5. Once the adversary has observed the outcome of the vote, which now should be a split up to an $O(\sqrt{W_{honest}})$ gap, the adversary uses its slot 2 committee members (which stipulates the adversarial fraction $O(1/\sqrt{W_{honest}})$ required for this attack) as well as slot 0 and 1 committee members to rebalance the vote to a tie. As the tie is restored, the adversary can use the same strategy in the following slot, and so forth. Note that the adversary can observe the outcome of a vote and learns how many honest committee members saw A and B leading. The adversary can use this information to improve its estimate of T_{delay}. The optimal T_{delay} can be reliably localized using grid search.

Conclusion

The future is ultimately unknowable, but unlikely to be secure. Ethereum is as of yet a fairly young technology, and one which exhibits a tremendous amount of complexity. Naturally, this promises a great variety of security vulnerabilities and potential future vulnerabilities, a selection of which have been explored here in the discussion of Layer 1 and Ethereum 2.0. The attractiveness of the EVM as a target for attack is enhanced by its direct operation on financial instruments, which offer both the reward and mechanism of attacks, as explored in our discussion of the security of distributed exchanges on Layer 2. Though many of these exploits can be avoided doctrinally, some remain unresolved, and some number will persist even after the refinement of Etherium has addressed those enumerated here.

Works Cited

Beekhuizen, C., 2020. *Validated, staking on eth2: #2 - Two ghosts in a trench coat* [online] Available at: <https://notes.ethereum.org/@vbuterin/lmd_ghost_mitigation> [Accessed 10 May 2022]

Buterin, V., 2017. *Casper the Friendly Finality Gadget.* [online] Available at: <https://arxiv.org/pdf/1710.09437.pdf> [Accessed 10 May 2022]

Deka, C., 2022. *Hackers Exploit Arbitrum-based Marketplace Treasure: Over 100 NFTs Stolen*. [online] CryptoPotato. Available at: <https://cryptopotato.com/hackes-exploit-arbitrum-based-marketplace-treasure-over-100-nfts-stolen/> [Accessed 10 May 2022].

Ethereum Improvement Proposals. 2022. *Ethereum Improvement Proposals*. [online] Available at: <https://eips.ethereum.org/> [Accessed 11 May 2022].

ethereum.org. 2022. *Layer 2*. [online] Available at: <https://ethereum.org/en/layer-2/> [Accessed 10 May 2022].

jakub, 2022. *Ethereum Layer 2 Scaling Explained*. [online] Finematics.com. Available at: <https://finematics.com/ethereum-layer-2-scaling-explained/#:~:text=Layer%202%20scaling%20can%20dramatically,2000%2D4000%20tx%2Fsecond.> [Accessed 10 May 2022].

Neu, J., 2022. *Two Attacks On Proof-of-Stake GHOST/Ethereum.* [online] Available at: <https://arxiv.org/pdf/2203.01315.pdf> [Accessed 10 May 2022]

Phillips, D., 2021. *What Is Arbitrum?*. [online] CoinMarketCap Alexandria. Available at: <https://coinmarketcap.com/alexandria/article/what-is-arbitrum> [Accessed 10 May 2022].

Reiff, N., 2022. *Polygon (MATIC)*. [online] Investopedia. Available at: <https://www.investopedia.com/polygon-matic-definition-5217569#:~:text=Polygon%20vs.-,Ethereum,as%20a%20blockchain%20development%20network.> [Accessed 10 May 2022].

Richards, S., 2022. *Optimistic Rollups*. [online] ethereum.org. Available at: <https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/> [Accessed 10 May 2022].

Sen, S., 2021. *Introduction to Ethereum Rollups explained - step-by-step beginners guides* | *QuickNode*. [online] Quicknode.com. Available at: <https://www.quicknode.com/guides/infrastructure/introduction-to-ethereum-rollups> [Accessed 10 May 2022].

Smith, C., 2022. *Scaling*. [online] ethereum.org. Available at: <https://ethereum.org/en/developers/docs/scaling/> [Accessed 10 May 2022].

Stevens, R., 2022. *What Is Arbitrum? Speeding Up Ethereum Using Optimistic Rollups*. [online] Decrypt. Available at: <https://decrypt.co/resources/what-is-arbitrum-speeding-up-ethereum-using-optimistic-rollups> [Accessed 10 May 2022].

Noyes, Charlie 2022. [online] Available at: <https://research.paradigm.xyz/MEV> [Accessed 11 May 2022].

OpenZeppelin Team, Docs.openzeppelin.com. 2022. *ERC20 - OpenZeppelin Docs*. [online] Available at: <https://docs.openzeppelin.com/contracts/4.x/erc20> [Accessed 11 May 2022].

Ens Domains Team, Ens.domains. 2022. [online] Available at: <https://ens.domains/> [Accessed 11 May 2022].

MEV-Explore, Explore.flashbots.net. 2022. *MEV Explore*. [online] Available at: <https://explore.flashbots.net/> [Accessed 11 May 2022].

Robinson, D. and Konstantopoulos, G., 2022. *Ethereum Is a Dark Forest*. [online] Medium. Available at: <https://medium.com/@danrobinson/ethereum-is-a-dark-forest-ecc5f0505dff> [Accessed 11 May 2022].