XDB[§]: On Mitigating Frontrunning attacks in Ethereum

Daniel Xu
danielxu@mit.edu

YeonHwan Park
parky@mit.edu

Xinyu Lin
linx3@mit.edu

Abstract— **The advent of new decentralized exchanges and applications to trade cryptocurrencies has abstracted away the technical knowledge the average user needs to know in order to trade different coins. Although this means that the average user will most likely never learn the technical underpinnings of the technology they are using, it does not mean that malicious adversaries will not. Aimed to decentralize markets and crown transparency in order to attract users, any cryptocurrencies have open-sourced their codebases and publicized their whitepapers. In this paper, we evaluate the structure of a popular cryptocurrency based on the popular blockchain technology, its security vulnerabilities, and how such a network might mitigate attacks against the users interacting with the network.**

1. Introduction

In October of 2008, an author by the pseudonym Satoshi Nakamoto published "Bitcoin: A Peer-to-Peer Electronic Cash System" [15] anonymously on a public mailing list dedicated to conversations about cryptography. Since then, the the concept of a blockchain has been has gained major traction in the research community and more than 18,465 new currencies have been created, along with thousands of exchanges aimed help users trade such assets. Decentralized exchanges, also known as DEXs, have abstracted away the technical know-how required from users and have enabled the general public to invest into a variety of different coins. While it is nowhere near necessary for the average user to be familiar with the inner technical workings of a cryptocurrency, this has meant that decentralized exchnages and cryptocurrencies have become the target for many malicious actors looking to take advantage of such users. By looking at open-sourced code and whitepapers published online, adversaries can easily exploit vulnerabilities in variety of networks.

In particular, we look at the technical structure of the Ethereum network and an a popular attack known as "frontrunning" user transactions. Frontrunning is a type of attack on any centralized or decentralized exchange where a malicious actor uses insider knowledge that others don't have to take advantage of potential trades and capitalize on the market [18]. While frontrunning is generally frowned upon and considered illicit in "normal" markets such as the stock exchange, it is legal in the cryptocurrency exchanges. This is because all of the knowledge for most given cryptocurrencies are public (a common mantra in the blockchain community is: "Don't trust, verify." This implies that anyone with sufficient technical knowledge should be able to reconstruct the entire protocol). While the knowledge known by adversaries in this model is not necessarily insider information because it is publicly available, most people who purchase and sell such currenices do not bother to learn the intricacies or implications of the technologies behind the cryptocurrencies, making them easy targets of such attacks.

2. Background

In order to understand the high-level frontrunning attacks on the users of the Ethereum network, we first dive into a brief overview of the intricacies behind the technology powering Ethereum known as the Blockchain, and then dive deeper into how different pieces are tied together.

[0]Short for "Xinyu, Daniel, and Brian" - note that we did not contribute any novel ideas to the field. We did, however, think it would be cool to title our paper this way.

2.1. Ethereum Overview

Launched in July of 2015 [8], Ethereum is a decentralized cryptocurrency based on the blockchain technology and has the second highest number of trades amongst different cryptocurrencies. Similarly to other popularized cryptocurrencies, it runs on blockchain technology and runs a proof of work model [8].

It is mostly similar to the popularized cryptocurrency Bitcoin as it uses a proof-of-work model [8] and is distributed and decentralized by blockchain technology [8]. However, it diverges from Bitcoin by presenting a new feature known as "smart contracts" [8] – in addition to a ledger functionality allowing users to trade Bitcoins, it also provides a decentralized way for users to run decentralized applications (also called dApps).

2.1.1. Blockchain

A blockchain is a type of linear datastructure similar in nature to a singly linked list [15]. In this analogy, each node is called a "block," and each pointer is a specialized link between blocks called "chains." The basic idea behind a blockchain is to connect a series of unchangeable nodes with self-verifiable pointers to provide an immutable, distributed, and decentralized datastructure with high levels of transparency. Given an established chain of blocks, any change in the datastructure should be easily observable. Given the full blockchain, any user should be able to connect a new set of nodes to extend the blockchain. Finally, there should be a generalized consensus algorithm to ensure that there is only one official version of the blockchain at any given time even as many different users may hold their own localized copies.

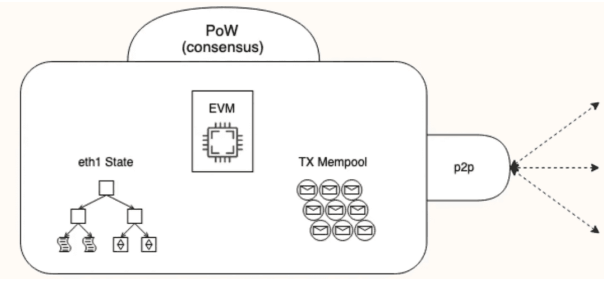

Fig. 1. Ethereum client diagram [8]

Many cryptocurrencies are built with the blockchain as their underlying architecture, and Ethereum is one example of such a network [8]. Ethereum uses its blocks to store verified transactions and chains the blocks together with an algorithm known as a "Proof of Work," also referenced as POW.

Proof of Work is one of many immutable and decentralized consensus schemes that can be used to power blockchain protocols. The overall simplified idea can be illustrated as follows:

1) The Ethereum network sets some small 256-bit value x.
2) Different clients gather transactions to package into the n_{th} block of the blockchain. Blocks consist of many transactions.
3) Clients compute many different hash values $Y = (y_1, ...y_n)$ until one finds a hash value y_i that is smaller than x. This client now has a "proof of work" that validates them to add the hash to the blockchain.

It is easy to see how setting x to be a small number such as a 256 bit number with 50 leading zeroes will make it difficult for any particular client to guess a valid number to "prove their work." There is $\frac{1}{2^{50}}$ chance of actually finding a solution in this case! The Ethereum network adjusts the value of x depending on the load on the network to ensure that transactions are processed at a predictable and steady rate.

In order to ensure that the blockchain as a whole is truly immutable, the hash of the previous block is added into the header of the next block [8]. While hashing the transactions themselves ensure that they stay in order (for instance, with a nonce – a counter that increments transaction orders), including the hash of the previous block into the next block ensures that the order of the blocks themselves are preserved. This is critical as it ensures that the state machine 3 keeping track of accounts and transactions stays up to date and malicious actors cannot reorder operations to incur an invalid state (e.g. Double Spend).

2.2. Ethereum Network

Ethereum maintains a distributed and decentralized state machine on the web by running on many

nodes, also known as clients 1, hosted around the world in order to ensure that the blockchain stays up and running. Each client runs its own "Ethereum Virtual Machine," also known as the EVM [8]. This is where the logic of transactions is materialized and the rules of the state machine are maintained. Code for smart contracts are written in the Solidity language which then gets translated into byte-source code to be run on the EVM. In addition, each client keeps track of its own copy of the blockchain and what is known as a "mempool" – the pool of uncommitted transactions each client has the choice to pull from [8].

In contrast to the traditional blockchain structure that bitcoin uses to keeps track of a financial ledger [15], Ethereum maintains a state machine to allow users to run decentralized applications on top of its blockchain by compiling code into what is known as a set of "smart contracts." Put simply, smart contracts are programs hosted on the Ethereum network that use the underlying blockchain as a database and the nodes around the world as compute resources.

Because all operations on the Ethereum network are processed by the EVM on some client, there needs to be an incentive that drives clients to process these operations. In the Ethereum protocol, each and every operation carried out on the network carries something called a "gas-price." Essentially, the gas price of an operation is a tax that each client has to pay in order to have the transaction eventually committed to the blockchain, part of which is then tipped to the client that added the block [8]. The gas price is adjusted based on [8]:

1) Job size: Larger jobs cost more gas
2) Job urgency: More urgent jobs cost more gas

The gas price (and as a result, the price of each Ether) fluctuates based on what is known as an "Automated Market Maker," which is talked about in more detail in sections below.

2.2.1. Life of a Transaction

To launch a transaction on the Ethereum network, a user is required to fill in some valid fields [8]:

- **Recipient** - The receiving address
- **signature** - Identifier of the sender
- **value** - The amount of ether being sent
- **data** - Optional field to include arbitrary data
- **gasLimit** - The maximum amount of gas units that cam be consumed by the transaction
- **maxPriorityFeePerGas** - The maximum amount of gas to be included as a tip to the miner
- **maxFeePerGas** - The maximum amount of gas willing to be paid for the transaction (inclusive of baseFeePerGas and maxPriorityForGas)

The "gas price" associated with a certain transaction is the amount of currency required to utilize the amount of compute resource that will be required to run the transaction on the Ethereum Network. The "gas price" incentivizes nodes on the ethereum network to pick up compute jobs for transactions in exchange for a portion of the gas price.

Once a transaction with valid fields is generated on the user's node, it is then transmitted to the rest of the network which then may or may not be picked up by nodes for further processing. In Ethereum, pending transactions live in a datastructure known as a "mempool." This is a place where transactions that have been submitted for processing but not yet been committed to the blockchain live. Mempools are an important feature of decentralized blockchains because transactions don't necessarily arrive at each compute node in order.

Because workers are incentivized with monetary rewards, transactions in the mempool are generally selected in preference for the jobs that have been submitted with a higher gas price. When a job is eventually picked up by a node, it then includes the transaction into a new block with other transactions to fill the block by hashing them into a Merkle tree. It then competes with other nodes that may have also picked up the job (also known as "miners" or "miner nodes") to generate a proof of work.

Once the block and a valid hash is successfully created, the client adds the block to the blockchain and broadcasts it to the rest of the network, solidifying the set of transactions included in it. In the case that two miners add to the chain at the same time, the network simply waits for the next block to be mined without a tie before confirming the new blocks [8].

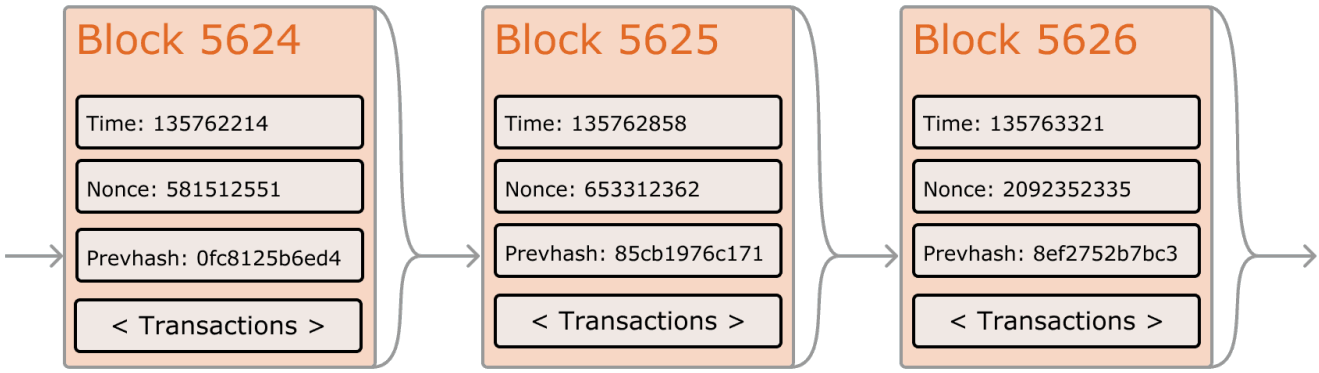

Fig. 2. Illustration of the underlying blockchain powering Ethereum [8]

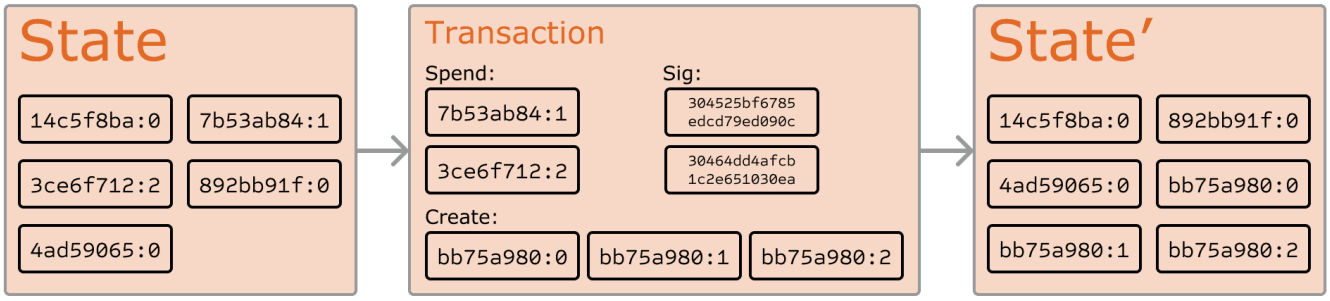

Fig. 3. Illustration of the underlying state machine powering Ethereum [8]

Frontrunning attacks take advantage of the priorities that Ethereum clients will naturally gravitate towards – given that most users will claim jobs with the highest gas price, how can malicious actors take advantage of the assumption and profit from the technical details of the Ethereum network?

3. Security Policies

In a traditional central exchange, there is a singular authority or a group of individuals who make administrative decisions on the behalf of the rest of the individuals using the service. However, decentralized exchanges differ significantly with regards to this view on security – there is not centralized group that makes decisions on the behalf of others. Instead, the responsibility of executing transactions and ensure they are secure is distributed amongst the users of the protocol through a unified consensus algorithm, i.e. a governance structure that ensures that everyone has a stake in the administration of the network. Even the slightest changes to the set of algorithms that governs the software requires a majority of the users to enact.

In our security policy, we look at two different categories of administrative networks: the decentralized exchange where the trade of certain cryptocurrencies occurs, and the cryptocurrency that is being traded.

3.1. Decentralized Exchanges

While decentralized exchanges are not part of the Ethereum Network directly, they oftentimes act as the middlemen for traders to buy and sell coins on the Ethereuem blockchain. While the Ethereum network and protocols allow users to buy and sell ethereum, it does not provide users for an automatic way to find such buyers for sellers and vice versa.

As such, decentralized exchanges should have the permissions to see open trades and match sellers selling at a set price with buyers who are willing to buy at that price or higher. On the other hand, decentralized exchanges should also be able

to match buyers with any sellers who are willing to sell for a certain price of lower. They should not be able to alter any part of the blockchain itself, nor edit the any transactions whether committed or uncommitted as to manipulate the trades between two individuals. The decentralized exchange should, however, be able to charge the buyer and/or the seller for fees for using its service.

3.2. Ethereum

The Ethereum network should have the permissions to read the blockchain until the latest uncommitted block. The Ethereum network (or individual clients) should have the permission to choose which transactions go into their new blocks and to hash values such that they can add onto the blockchain. However, they should not be able to commit to the blockchain if they are given an updated blockchain with newer transactions (they would have to restart work).

3.2.1. Traders

In this security model, buyers and sellers should have the same permissions. Buyers should be able to purchase Ether for the agreed upon price, and sellers should have the permission to sell the blockchain in their ownership to others for a set price. In this security model, neither the buyer nor the seller should be surprised by the amount of Ether they end up selling or receiving after the transaction is complete.

In addition, buyers and sellers should not be able to manipulate the blockchain or any of the smart contracts built into it if they don't own them.

4. Automated Market Maker

Automated market makers, AMM, are the underlying protocols that powers a decentralized exchange. It allows assets to be traded by using crypto liquidity pools as counterparties, instead of a traditional market of buyers and sellers. Instead of order books, AMMs use algorithms to match buyers and sellers.

In particular, AMM use formulas to allow trading token pairs. For example, Bitcoin-USDollar

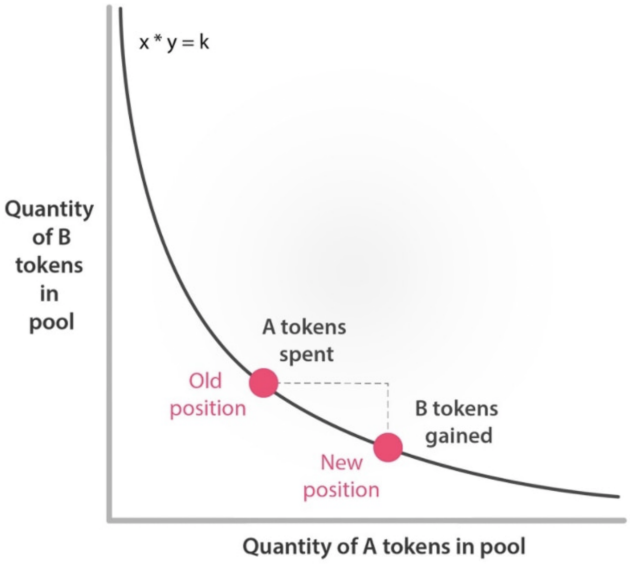

Fig. 4. Product rule states that the product of two liquiditiy stays constant for the contract [17].

is a trading token pair. These formulas are also known as smart contracts. AMMs do not use order books or order types, instead, the formulas determine asset prices. Because of AMMs, we do not need a third party or trader to make a trade. Instead, individual traders interact directly with the contract, which makes the market for you.

One popular rule many AMMs used today is the constant product rule, which states that total amount of liquidity remains constant. As shown in Figure 4, the product of the number of A token and number of B token must be a constant.

4.1. Example of AMM with insertion attack

In an insertion attack, also known as sandwich attack, the attacker places one transaction before and one transaction after the victim's transaction in hope of making some profit (refer to Section 5.2.) for more details. In this section, we examine how an AMM works and allows for insertion attacks.

Assume our liquidity pool contains 1000 A tokens and 100 ETH. The product of the two quantities is 100000, and this value must stay constant throughout. Thus, the current price of A is 0.1 ETH. In this scenario, let Bob be the attacker and Alice be the victim. Assume that Bob has

some prior information about Alice's transaction.

1) Bob makes a pre-emptive trade. He spends 5 ETH to buy A before Alice's transaction. The product rule states the following:

$$(1000 - X) * (100 + 5) = 100000$$

X is the number of A Bob is able to purchase with 5 ETH. Note that we start with 1000 A and 5 ETH. Solving the equation above reveals that $X = 47.62$. As a result, **Bob buys 47.62 A for 5 ETH**.

2) Next would be Alice's normal transaction. She spends 10 ETH to buy A. Note that Bob had knowledge of this transaction, so he was able to buy A beforehand. After Bob's transaction, there are 952.38 A and 105 ETH in the liquidity pool. The product rule states:

$$(952.38 - X) * (105 + 10) = 100000$$

Solving the equation above reveals that $X = 82.81$. Thus, after the transaction is executed, **Alice buys 82.81 A for 10 ETH**.

3) Finally, Bob sells the 47.62 A he initially bought. After Alice's transaction, there are 869.57 A and 115 ETH in the liquidity pool. The product rule states:

$$(869.57 + 47.62) * (115 - Y) = 100000$$

Solving the equation above reveals that $Y = 5.97$. Thus, after the transaction is executed, **Bob sells 47.62 A for 5.97 ETH**.

With this insertion attack, **Bob makes a net gain of 0.97 ETH**. This example illustrates the role AMM in pricing the assets in the liqudity pool of a decentralized exchange.

5. Frontrunning types

All transactions are visible in the mempool for a short period of time before being executed, thus, observers of the network can see and react to an action before it is included in a block. For example, in a decentralized exchange where a buy order transaction be can seen, a second order can be broadcast and executed before the first transaction is included. There are three main categories of front-running attacks: displacement, insertion, and suppression (Figure 5).

5.1. Displacement

In a displacement attack, an attacker A observes a profitable transaction T_V from victim V and broadcasts its own transaction T_A to the network. T_A has a higher gas price than T_V, and miners will include T_A before T_V. In a displacement attack, the attacker does not require the victim's transaction to execute successfully. For example, consider a scenario where a smart contract awards a user with a prize if they can guess the preimage of a hash. The attacker can wait for a user to find the solution, copies the solution, and submit to the network. In this case, the attacker's transaction will be mined first, thereby winning the prize, and the user's transaction will be mined last.

5.2. Insertion

In an insertion attack, an attacker A observed a profitable transaction T_V from a victim V. The attacker then broadcasts two transactions T_{A1} and T_{A2} to the network such that T_{A1} has a higher gas price than T_V and T_{A2} has a lower gas price than T_V. Insertion attacks are also known as sandwich attacks. In insertion attacks, T_{A2} depends on the successful execution of transaction T_V. A well-known example of insertion attacks is arbitraging on decentralized exchanges, where an attack observes a large trade, also know as a whale, sends a buy transaction before the trade to drive up the value of the contract, and then followed by a sell transaction.

5.3. Suppression

In a suppression attack, an attacker A observes a transaction T_V from a victim V and broadcasts its transactions to the network. The attacker's transactions have a higher gas price than T_V so that miners will include A's transactions before T_V. By doing so, A will be able to suppress transaction T_V by filling up the block with its transactions so that T_V cannot be included anymore in the next block. Suppression attacks are also known as block stuffing. Each block in Ethereum has a block gas limit. The consumed gas of all transactions included in a block cannot exceed this limit. A's transactions will try to consume as much gas as possible to

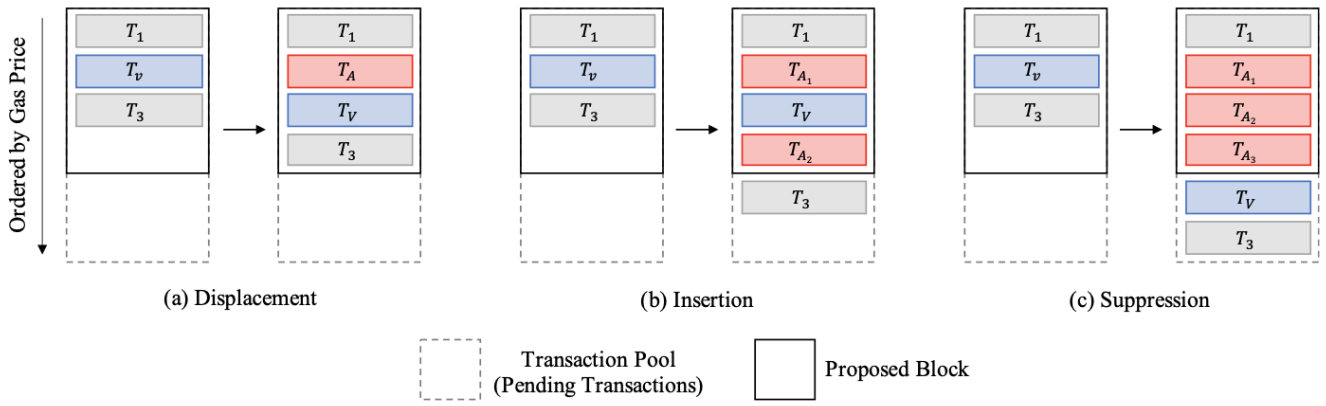

Fig. 5. Illustration of three main categories of frontrunning attacls: displacement, insertion, and suppression. Blue blocks T_V represent victims' transactions, and red blocks T_A represent the attacker's transactions [9]

reach this limit so that no other transaction can be included in the block. Suppression attacks are often used in lotteries where the last purchaser of a ticket wins if no one else purchase a ticket during a specific time window. Attackers can purpose a ticker and mount a suppression attack for several blocks to prevent other users from purchasing the ticket. Note that type of frontrunning attack is expensive as it requires the attacker to place multiple transactions with higher gas price than the victim's transaction T_V.

6. Mitigating Attacks

There are a number of ways to prevent, detect, and mitigate frontrunning attacks. However, the details of exact solutions change over time and depends on the decentralized exchange. In this section, we extracted the main principles that address the attack and analyzed the advantages and drawbacks of each. A particular system may implement more than one.

Most mitigations of front-running attacks can be categorized into three classes. In the first class, the blockchain removes the miner's ability to arbitrarily order transactions by enforcing some ordering, or queue, for the transactions. In the second class, cryptographic techniques are used to limit the visibility of transactions, giving the potential frontrunning less information to base their strategy on. In the final class, decentralized applications and smart contracts are designed from bottom-up to remove the importance of transaction ordering or time in their operations.

6.1. Transaction Sequencing

Ethereum miners store pending transactions in pools and draw from them when forming blocks. There is no intrinsic order to how transactions are drawn, so miners are free to sequence them arbitrarily. The vanilla Go-Ethereum implementation prioritizes transactions based on their gas price. Since no rule is enforced, miners can sequence transactions in advantageous ways. This created opportunities for displacement, insertion, and suppression attacks, which all depend on the ordering the transactions.

However, a first-in-first-out order (FIFO) is not possible on a distributed network because transactions can reach different nodes in a different order. Although the network can form a consensus based on locally observed FIFO, this would increase the rate of orphaned blocks, add complexity to the protocol, and be unfair to certain users. We can use a trusted third party to assign sequential numbers to transactions, but this will be contrary to blockchain's core innovation of distributed trust. However, some exchanges such as EtherDelta and OxProject do centralize time-sensitive functionalities.

An alternative would be to sequence transactions pseduoradomly. For example, propposals like Canonicial Transaction Ordering Rule by Bitcoin

Cash ABC [11] adds transactions in lexicographical order according to their hash. Ethereum can apply this rule to make frontrunning statistically difficult, but the protection is marginal at best and might even open up doors for more attacks. For example, front-runner can construct multiple transactions, with slightly different values, until they find a transaction lands at an optimal position on the sequence. The front runner broadcasts only this transaction. The miners that include this transaction will position it in front of transactions they heard about much earlier.

Another potential solution is having transactions themselves enforce order. For example, transactions can specify the current state of the contract as the only state to execute on. This transaction chaining only prevents insertion attacks but not displacement attacks. A transaction chaining only allows one state-changing transaction per state, which is a drawback for active decentralized applications.

6.2. Confidentiality

6.2.1. Privacy-Preserving Blockchain

All transaction details in Bitcoin are made public and participant identities are only lightly protected. A number of techniques increase confidentiality and anonymity for crytocurrencies. Many might believe that a confidential decentralized application would not permit frontrunning since the front-runner does not have access to the details of the transaction he is front-running. However, they are some edge cases to explore.

A decentralized-application interaction includes the following components:

1) the code of the decentralized application
2) the current state of the decentralized application
3) the name of the function being invoked
4) the parameters supplied to the function
5) the address of the contract the function is being invoked on
6) the identity of the sender

Confidentiality applied to a decentralized application could mean different levels of protection for each of these. For frontrunning, function calls (3,4) are the most important. However, function calls could also be inferred from state changes.

The use case of privacy-preserving blockchains need to be evaluated on a case-by-case basis. For instance, one method used by traditional financial exchanges to mitigate frontrunning attacks from high frequency traders is a dark pool. Dark pool is a (2,3,4)-confidential order book maintained by a trusted party. Decentralized applications and smart contracts can replace this trusted party. If the contract addresses are known (no 5-confidentiality), frontrunners, frontrunners can know about the traffic pattern of calls to contracts. This opens up many opportunities for attacks. For example, if each asset on an exchange has its own market contract, this leaks trade volume information.

6.2.2. Commit/Reveal

Although confidentiality alone is insufficient for mitigating frontrunning attacks, a hybrid approach of sequencing and confidentiality can be effective. We can apply a crytographic trick known as commit/reveal. The main idea of commit/reveal is to protect the function call until the function is queued in a sequence of functions to be executed. Once the sequence is established, the confidentiality is lifted and the function can only be executed in the order it was queued.

A commitment scheme enables one to commit to a digital value while keeping it a secret to the public. The committed value can then be opened at a later time when the committer wills. One common use case of this scheme is to submit the cryptographic hash of the value to a smart contract, and later reveal the original value, which can be verified by the contract to correctly hash to the commitment. Hashing the initial value reduces the risk of frontrunning attacks since attackers cannot see the values.

One application of this scheme to blockchain is Namecoin [12], is a cryptocurrency originally forked from bitcoin software. Namecoin adopts the commit/reveal scheme because users send a committ transaction which registers a new hidden domain name. Once the first transaction is confirmed, a time delay begins. After the delay, a second transaction reveals the details of the requested domain. If the reveal transaction is executed and

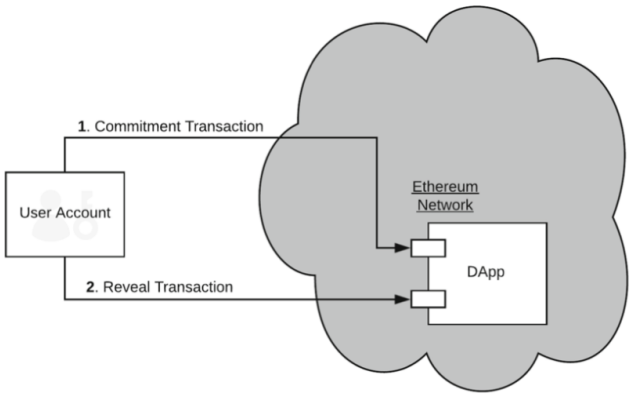

Fig. 6. In the commit/reveal scheme, the user first commits a hashed transaction. When the transaction is ready to be executed, the user then sends a key to unlock the hashed transaction [9].

confirmed faster than an adversarial transaction, the attack can be prevented.

Note that commit/reveal is a two step process, and aborting after the first round could be an issues. For example, the attacker can spray the sequence with multiple committed transactions with no intention of executing them at all. Also note that any muti-step protocol can be unstable on a distributed network.

6.3. Design Practice

The last category of mitigation is to assume that front-running is unpreventable and we must redesign the functionality of decentralized applications to remove any benefit from it. For example, decentralized exchanges can adopt a call market design instead of a time-sensitive order book to disincentivize frontrunning. In a call market design, prices are dictated by the exchange rather than by bids and offers. Orders are also aggregated and collected at designated intervals instead of trading throughout the day.

Malinova et al. [14] also discussed front-running mitigations for blockchain-based trading platforms. Instead of investigating decentralized applications, they developed an economic model where transactions, asset holdings, and traders' identities have greater transparency than in standard economic models. They argued that transparency can be achieved by blockchain technology. However, in their model, they assumed that entities can interact directly over private channels to arrange trades. They defined front-running in the context of private offers, where parties might adjust their position before accepting or countering an offer. This model is different from the decentralized models where entities cannot interact with one another over private channels.

7. ZK-Snarks

Perhaps the most secure way of preventing front running is to hide certain details of the transaction altogether. Zero Knowledge proofs are proofs where the prover can prove knowledge of a value without revealing the value to the verifier[10]. A critical part of the sandwich attack is that the adversary can take advantage of the Automated Market Maker to generate profit. Recall, after each transaction, the product of the quantities of both assets in the liquidity pool must be kept constant. Thus, the values of both quantities determines what the relative price of the assets are. Using Zero Knowledge proofs, a trader could hide the exact quantities of each asset and submit a proof that the product of the quantities is constant! [1] The exact quantities would be revealed at a later time. This discourage sandwich attacks as the adversary can no longer calculate the exact value of each transaction and cannot calculate when trades are profitable.

A simple example of a Zero Knowledge proof is the following: Given a public value g^x where x is secret, the prover wishes to prove knowledge of x. The prover and verifier performs the following steps:

1) Prover picks value v and sends g^v to the Verifier.
2) Verifier picks value c and sends it to the Prover
3) The Prover calculates $v - xc$ and sends it to the Verifier
4) The Verifier evaluates $g^{v-xc} * (g^x)^c$ and verifies if this value is equal to g^v.

This process can be repeated many times until the verifier is properly convinced that the prover knows x[2].

The ZK-Snark (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) takes the zero

knowledge proof a step further and enforces that the verifier does not need to interact with the prover [7]. That is, the verifier and the prover do not have to be online at the same time. The prover can publish a proof and the verifier can read it and verify it at a later date. This allows for zero knowledge proofs to be implemented in a much wider range of practical applications. For example, it would be very difficult if every Ethereum transaction required multiple parties to be online at the same time and verify the transactions using an interactive zero knowledge proof.

We can convert the above example into a ZK-Snark by enforcing that the value of c is some deterministic Hash function of g, g^x, g^v! If the Hash function is secure, the prover cannot control the value of c and thus it will be hard to fake the proof. This removes the need for the verifier to be present during the proof. Instead, any verifier can read the published proof and verify for themselves that this works. While this example seems quite simple, in reality, ZK-Snarks are quite complicated.

To be practically used, the ZK-Snark must satisfy some strict properties.

- **Succinctness** - The size of the published proof should be small, regardless of the amount of computation required to obtain the value being proved.
- **Non-Interactivity** - The prover must be able to publish a proof that can be verified by any verifier without direct interaction with the prover.
- **Correctness** - The statement being proved must be correct with very high probability.
- **Zero Knowledge** - The verifiers should not be able to extract any information about the information being proved.

In the following, we will provide an outline of some of the fundamental underlying concepts of ZK-Snarks, including a description of the Pinocchio protocol, but will stop short of providing a discussion for how ZK-Snarks are implemented in practice.

7.1. Pinocchio Protocol

In this section, we present the Pinocchio protocol[16], the first protocol to create general ZK-Snarks which are efficient enough to be used at scale. The Pinocchio protocol in essence allows the prover to prove that they performed a series of computations and have calculated a valid solution without actually revealing any information on the solution itself! We will walk through the key ideas behind the Pinocchio protocol and finally conclude with a full discussion on how the Pinocchio protocol works.

7.1.1. The Polynomial Method

The core idea behind many ZK-Snarks is exploiting the fact that the number of roots of a polynomial is upper bounded by the degree of the polynomial. Given two distinct polynomials of degree d, they can be equal in at most d points. For a fixed value s the fact that two polynomials f and h satisfy the property $f(s) = h(s)$ is negligibly small unless $f = h$. Thus, if the prover can show the evaluation of f, h is the same at some point s outside of the prover's control, then the verifier can safely assume $f = h$ [5].

Polynomials exhibit many nice properties which we will discuss more in detail in the following sections. For example, given $d + 1$ points, we can efficiently construct the unique degree d polynomial that passes through all of those points using the Lagrange basis polynomials.

In the following discussion, we will ignore details about the field we do computations in. In implementation, these polynomials live in finite fields, as well as any computations we perform.

7.1.2. Evaluation at a Hidden Point

The next piece of the polynomial method requires the prover evaluating a polynomial, in the encrypted space, at a point s without the prover being able to discover s. Recall, the encrypted space is a mapping $x \to g^x$. The Diffie-Hellman Assumption provides confidence that it is hard to recover information about the original value from the encrypted value. To start, the verifier will provide the following values: $g, g^s, g^{s^2}, ..., g^{s^d}$. The prover can then calculate $g^{p(s)}$ for any given polynomial $p(x) = a_d x^d + ... + a_0$ by the formula $g^{p(s)} = (g^{s^d})^{a_d} * (g^{s^{d-1}})^{a_{d-1}} * ... * g^{a_0}$. Thus, the prover has evaluated the polynomial at a hidden

point s and can publish this value without revealing information about the polynomial coefficients [5].

7.1.3. Knowledge of Exponent Assumption

Suppose the prover is given the values (g, g^a) where a is secret. The Knowledge of Exponent Assumption assumes that if the prover has calculated g^{ar} for some number r, the prover must have done the computation $(g^a)^r$, i.e. raised the value g^a to the power of r rather than raising g^r to the power of a. Thus, if the prover can publish values X, Y where $X^a = Y$, then since the prover does not know a, and using Knowledge of Exponent Assumption, this guarantees the prover must have calculated X, Y by raising each of g, g^a to the same power r [6].

We can generalize this framework to multiple pairs of values. If the prover is given values $(g_1, g_1^a), ..., (g_k, g_k^a)$, and the prover publishes values X, Y where $X^a = Y$, then there must be values $w_1, ..., w_k$ where the prover calculated X, Y using the following computations:

$$X = g_1^{w_1} ... g_k^{w_k}$$
$$Y = (g_1^a)^{w_1} ... (g_k^a)^{w_k}$$

Thus, we can reason that the value X came from multiplying some combination of the g_i's. Furthermore, X, Y come from the exact same coefficients w_i! This idea is very powerful and is at the heart of the Pinocchio protocol.

7.1.4. Pairing

In many use cases of Cryptography were we consider values in groups, we choose groups which we hope to satisfy the Decisional Diffie-Hellman Assumption. That is, given three values x, y, z, it is impossible to tell whether these numbers are of the form g^a, g^b, g^{ab}. We will assume without proof that there exists groups G_1, G_2, G_T and bilinear function $e : G_1 \times G_2 \mapsto G_T$ [13] that satisfy for all group elements g_1, g_2, g_3 in the aforementioned groups, and integers a, b, the following requirements hold:

$$e(g_1^a, g_2) = e(g_1, g_2)^a$$
$$e(g_1, g_2^b) = e(g_1, g_2)^b$$
$$e(g_1 g_2, g_3) = e(g_1, g_3) e(g_2, g_3)$$
$$e(g_1, g_2 g_3) = e(g_1, g_2) e(g_1, g_3)$$

This obviously breaks the DDH assumption as we can always check if $e(g^a, g^b) = e(g^{ab}, g)$. However, this lets us verify the product of certain exponents $a * b = ab$ without revealing a, b itself. The groups G_1, G_2, G_t and the function e are beyond the scope of this paper. For further discussion, refer to the field of Elliptic Curve Cryptography.

7.1.5. Parameter Generation

One of the end goals of ZK-Snarks is to be non-interactive. This means that we should not require the verifier to generate values for the prover. This leads us to the question, how are the values $(g_1, g_1^a), ..., (g_k, g_k^a)$ and $g, g^s, g^{s^2}, ..., g^{s^d}$ created? How should they be used? If these values must be recomputed for each prover and verifier pair, then this would destroy the non-interactivity condition. Ideally, these values could be used by the prover (or multiple provers) and any verifier could verify the proof. If any one person knows the value of a, s, then they could create false proofs.

Given a trusted party, the trusted party could generate $(g_1, g_1^a), ..., (g_k, g_k^a)$ and $g, g^s, g^{s^2}, ..., g^{s^d}$ and then throw away a, s. These public values are known as public parameters. The values a, s are refered to as "toxic waste" and cannot be known by anyone. The current method for generating public parameters is by using elaborate generation "ceremonies" where multiple parties must stay online and interact with each other to generate these parameters without any one part knowing the entirety of the parameters. The exact details of these ceremonies are beyond the scope of this paper. ZCash has successfully implemented these ceremonies to generate public parameters for the ZK-Snarks used for ZCash transactions[3].

7.1.6. Encoding of Computation

We now begin discussion on the actual Pinocchio Framework. In this framework, the prover wishes to prove that they did a certain series of

computations [16]. The computations can involve any number of variables and can involve addition, subtraction, multiplication, and exponentiation by a constant integer value. For example, a simple computation can be something like $y = x^3$, $z = x + y + 3$.

The computations are encoded as a series of equations in a very specific form. We will "unroll" the computations into n variables and n equations. Suppose the variables are $w_1, ..., w_n$. By default the first equation is $w_1 = 1$. The kth equation is of the following form:

$$w_k = (a_{1,k}w_1 + ... + a_{k-1,k}w_k)(b_{1,k}w_1 + ... + b_{k-1,k}w_k)$$

For example, our simple computation can be unrolled into the following sequence of equations:

1) $w_1 = 1$
2) $x = k$
3) $T1 = x * x$
4) $y = T1 * x$
5) $z = x + y + 3w_1$

And our solution vector $W = \langle 1, x, T1, y, z\rangle = \langle 1, k, k^2, k^3, k^3 + k + 3\rangle$ for some value of k.

7.1.7. Rank One Constraint System

If we have some vector W which satisfies all of the n equations, then W must be the result of computing the computation! Thus, the Prover wishes to prove that they know a valid vector W which satisfies each equation. note that the coefficients of the equations $a_{i,j}$ and $b_{i,j}$ are public and they uniquely define the computation [4].

The equation can be written as:

$$C_k \cdot W = (A_k \cdot W)(B_k \cdot W)$$

Where C_k is the kth basis vector and $A_k = \langle a_{1,k}, ..., a_{k-1,k}, 0, ..., 0\rangle$ and $B_k = \langle b_{1,k}, ..., b_{k-1,k}, 0, ..., 0\rangle$.

These equations collectively are known as the Rank One Constraint System (R1CS).

7.1.8. Quadratic Arithmetic Program

The $3n$ vectors A_k, B_k, C_k (for all k) are publicly available.

We wish to transform the constants of the n equations into polynomials. Using Lagrange Interpolation, we can creat $3n$ degree $n-1$ polynomials $A_i(x), B_i(x), C_i(x)$ (for all i) where $A_i(k) = a_{i,k}$, $B_i(k) = b_{i,k}$, $C_i(k) = c_{i,k}$ for all i, k. Notice that these $3n$ polynomials are uniquely determined by the constraints of the system. We assume that these polynomials are publicly available. Thus, our n equations can be rewritten as:

$$\sum_i w_i C_i(k) = (\sum_i w_i B_i(k))(\sum_i w_i B_i(k))$$

for all integer $1 \leq k \leq n$.

Now, let's consider the three polynomials $A(x) = \sum_i w_i A_i(x)$, $B(x) = \sum_i w_i B_i(x)$, $C(x) = \sum_i w_i C_i(x)$. Then we have that $A(x)B(x) - C(x)$ is divisible by $Z(x) = (x - 1)(x - 2)...(x - k)$ The polynomials A, B, C and vector W. comprise the Quadratic Arithmetic Program (QAP) [5]! These polynomials are the basis of the Zero Knowledge proof. The Zero Knowledge proof consists of proofs for the following three statements:

1) The polynomials A, B, C must be some linear combination of $A_i(x), B_i(x), C_i(x)$
2) The polynomials A, B, C must actually be the same linear combination of $A_i(x), B_i(x), C_i(x)$. That is, there is some vector W where

$$A(x) = W \cdot \langle A_1(x), ..., A_n(x)\rangle$$

$$B(x) = W \cdot \langle B_1(x), ..., B_n(x)\rangle$$

$$C(x) = W \cdot \langle C_1(x), ..., C_n(x)\rangle$$

3) The polynomial $A(x)B(x) - C(x)$ is divisible by $Z(x) = (x - 1)(x - 2)...(x - n)$.

7.1.9. Additional Public Parameters

In order for the prover to prove these statements, additional public parameters must be available. The public parameters are as follows: The secret "toxic waste" variables are k_a, k_b, k_c, k, t. For $1 \leq i \leq n$, the following values are the public parameters[4]:

$$(g^{A_i(t)}, g^{k_a A_i(t)})$$

$$(g^{B_i(t)}, g^{k_b B_i(t)})$$
$$(g^{C_i(t)}, g^{k_c C_i(t)})$$
$$(g^{A_i(t)+B_i(t)+C_i(t)}, g^{k(A_i(t)+B_i(t)+C_i(t))})$$
$$g^{k_a}, g^{k_b}, g^{k_c}, g^{k}$$

Note that if anyone knows the "toxic waste" variables, they could construct falsified proofs!

The prover should be able to publish 8 values $X_a, Y_a, X_b, Y_b, X_c, Z_c, X, Y$ where $X_a^{k_a} = Y_a$, $X_b^{k_b} = Y_b$, $X_c^{k_c} = Y_c$, $X^k = Y$, and $X_a X_b X_c = X$.

An honest prover can generate these values by evaluating:

$$X_a = g^{A(t)} = \prod_i (g^{A_i(t)})^{w_i}$$
$$Y_a = g^{k_a A(t)} = \prod_i (g^{k_a A_i(t)})^{w_i}$$
$$X_b = g^{B(t)} = \prod_i (g^{B_i(t)})^{w_i}$$
$$Y_b = g^{k_b B(t)} = \prod_i (g^{k_b B_i(t)})^{w_i}$$
$$X_c = g^{C(t)} = \prod_i (g^{C_i(t)})^{w_i}$$
$$Y_c = g^{k_c C(t)} = \prod_i (g^{k_c C_i(t)})^{w_i}$$
$$X = g^{A(t)+B(t)+C(t)} = \prod_i (g^{A_i(t)+B_i(t)+C_i(t)})^{w_i}$$
$$Y = g^{k(A(t)+B(t)+C(t))} = \prod_i (g^{k(A_i(t)+B_i(t)+C_i(t))})^{w_i}$$

The verifier can verify these values by using the pairing function e which we assumes exists on the group. Note that according to our previous discussion, the verified can verify these equations without knowing k_a, k_b, k_c, k due to the properties of the pairing function!

If these equations are verified, then the verifier can be confident in each of the statements needed to complete the proof! From our discussion on Knowledge of Exponent Assumption, because $X_a^{k_a} = Y_a$, then X_a must be some product of the values $g^{A_i(t)}$. Thus we know that $X_a = \prod_i (g^{A_i(t)})^{w_i^a}$ for some vector w^a. The verifier doesn't know w^a but is confident that this vector exists. Now let $A'(x) = \sum_i w_i^a A_i(x)$. Similarly define vectors w^b, w^c and polynomials $B'(x), C'(x)$. Now, we need to verify the fact that A', B', C' were generated from the same vector w! Look at X, Y. Since $X^k = Y$, the verifier is confident that there is some vector w such that $X = \prod_i (g^{(A(t)+B(t)+C(t))})_i^w$. If $X_a X_b X_c = X$, then we know

$$A'(t)+B'(t)+C'(t) = \sum_i w_i^a A_i(t)+w_i^b B_i(t)+w_i^c C_i(t)$$
$$= \sum_i w_i A_i(t) + w_i B_i(t) + w_i C_i(t)$$

Since t is secret, the verifier can be confident that the polynomials on both sides of the equation are in fact the same polynomial! Note that all this time the verifier does not have any information on w, w^a, w^b, w^c! Furthermore, since the polynomials are equal, we can also be confident that $w = w^a = w^b = w^c$.

The final step in the proof is to verify that there exists some polynomial $H(x)$ where $A(x)B(x) - C(x) = H(x)Z(x)$. The prover actually publishes a ninth value $g^{H(t)}$. The final verification step is to verify the statement

$$e(g^{A(t)}, g^{B(t)})/e(g^{C(t)}, g) = e(g^H(t), g^Z(t))$$

With all these facts verified, the verifier can be confident that the prover has a valid vector w which satisfies the computation constraints!

References

[1] Amm front-running resistance with snarks.

[2] Breaking down eth 2.0 – zk-snarks and zk-rollups.

[3] Parameter generation.

[4] Zk-snarks: Under the hood.

[5] Aritra Banerjee, Michael Clear, and Hitesh Tewari. Demystifying the role of zk-snarks in zcash. In *2020 IEEE Conference on Application, Information and Network Security (AINS)*, pages 12–19. IEEE, 2020.

[6] Mihir Bellare and Adriana Palacio. The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In *Annual International Cryptology Conference*, pages 273–289. Springer, 2004.

[7] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct {Non-Interactive} zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, 2014.

[8] Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2013.

[9] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. 2019.

[10] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, 1994.

[11] Shammah Chancellor (Bitcoin ABC) Tomas van der Wansem (Bitcrust) Joannes Vermorel (Lokad), Amaury Séchet (Bitcoin ABC). Canonical transaction ordering for bitcoin. 2018.

[12] Harry A. Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. An empirical study of namecoin and lessons for decentralized namespace design. In *WEIS*, 2015.

[13] Neal Koblitz and Alfred Menezes. Pairing-based cryptography at high security levels. In *IMA International Conference on Cryptography and Coding*, pages 13–36. Springer, 2005.

[14] Katya Malinova and Andreas Park. Market design for trading with blockchain technology . 2016.

[15] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf.

[16] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE, 2013.

[17] Yongge Wang. Automated market makers for decentralized finance (defi), 2020.

[18] wordreference. Frontrunning.