High-Frequency Trading on Decentralized On-Chain Exchanges

Liyi Zhou *, Kaishua Qin *, Christof Ferreira Torres †, Due V Le ‡ and Arthur Gervais *
*University of Luxembourg, Luxembourg
†University of Luxembourg, Luxembourg
‡Purdue University, United States

Abstract—Decentralized exchanges (DEXs) allow parties to participate in financial markets while retaining full custody of their funds. However, the transparency of blockchain-based DEX in combination with the latency for transactions to be processed, makes market-manipulation feasible. For instance, adversaries could perform front-running — the practice of exploiting (typically non-public) information that may change the price of an asset for financial gain.

In this work we formalize, analytically expositor and empirically evaluate an augmented variant of front-running: sandwich attacks, which involve front- and back-running victim transactions on a blockchain-based DEX. We quantify the probability of an adversarial trader being able to undertake the attack, based on the relative positioning of a transaction within a blockchain block. We find that a single adversarial trader can earn a daily revenue of over several thousand USD when performing sandwich attacks on one particular DEX — Uniswap, an exchange with over 5M USD daily trading volume by June 2020. In addition to a single-adversary game, we simulate the outcome of sandwich attacks under multiple competing adversaries, to account for the real-world trading environment.

I. INTRODUCTION

Decades of asset trading on traditional exchanges have brought to fruition a veritable collection of market-manipulation techniques, such as front-running [6], pump and dump schemes [54] and wash trading [3]. In the context of cryptocurrencies, research to date indicates that the ecosystem requires a greater awareness of such malpractices [54], [21], [45], and better exchange design [12] to prevent misbehavior. Most existing legislation does not regulate crypto-exchanges as traditional exchanges — leaving ignorant traders open to exploitation by predatory practices, some of which is close to risk-free.

Decentralized exchanges (DEXs) allow traders to trade financial assets without giving up asset custody to a third party. Orders can be placed and matched in their entirety through immutable blockchain smart contracts, offering the possibility of censorship resistance, where orders cannot be modified prior and after execution.

Disclosure: Arthur Gervais works on the Liquidity Network, a community-driven, open source layer-2 blockchain scaling solution.

Censorship-resistant trade is itself made possible through reliance on an underlying blockchain, which makes public all attempted and executed trades within its peer-to-peer (P2P) network. The transparency of the blockchain layer, however, in combination with the latency for orders to deterministically execute makes, front-running easier to undertake — and hence influences negatively the security of the trader’s assets.

This paper. We focus on a combination of front- and back-running, known as a sandwiching, for a single on-chain DEX. To the best of our knowledge, we are the first to formalize and quantify sandwich attacks. To make their sandwich, a predatory trader first observes a blockchain P2P network for a victim transaction and then rushes to squeeze it by placing one order just after the transaction (i.e. front-run) and one order just after it (i.e. back-run). If the target transaction is going to increase (decrease) the price of an asset, the adversary can place an order before which buys (sells) the asset in question, and an order afterward which sells (buys) the asset again.

We restrict our focus to automated market maker (AMM) DEXs [52], [4], as opposed to DEXs which operate limit order books (LOB) [20], on account of their deterministic nature which enables us to rely on fewer assumptions in our analysis. AMM DEXs simplify trading by algorithmically performing market making, resulting in near-instant liquidity (i.e. the ability to purchase and sell assets) for market participants. Uniswap is a prominent example of an AMM DEX, which, by March 2020, has amassed a total liquidity of nearly 48M USD (corresponding to a 75% market liquidity share for AMM DEX) and had a trading volume of over 250M USD since its inception in November 2018. We formalize, analytically expositor and empirically evaluate sandwiching on AMM DEXs. We quantify optimal adversarial revenues and perform a real-world empirical evaluation of sandwich attacks. We also study the probability of a transaction having a particular relative position within a blockchain block, informing the prospects for such an attack. Finally, to account for a real-world scenario in which multiple adversaries are likely to

*While the SEC defines front-running as an action on private information, we only operate on public trade information.

The process of serving a market with the possibility to purchase and sell an asset.
to quantify the transaction fees resulting from a reactive fee counter-bidding contest.

Summary of contributions:

- Formulation of sandwich attacks. We state a mathematical formalization of the AMM mechanism and the sandwich attack, providing an adversary with a framework to manage their portfolio of assets and maximize the profitability of the attack.
- Analytic and empirical evaluation. We analytically and empirically evaluate sandwich attacks on AMM DEX. Besides an adversarial liquidity taker, we introduce a new class of sandwich attacks performed by an adversarial liquidity provider. We quantify the optimal adversarial revenue and validate our results on the Uniswap exchange (largest DEX, with 5M USD trading volume at the time of writing). Our empirical results show that an adversary can achieve an average daily revenue of 3,414 USD\(^4\). Even without collusion with a miner, we find that, in the absence of other adversaries, the likelihood to position a transaction before or after another transaction within a blockchain block is at least 79%, using a transaction fee payment strategy of ±1 Wei\(^5\).
- Multiple Attacker Game. We simulate the sandwich attacks under multiple simultaneous attackers that follow a reactive counter-bidding strategy [21]. We find that the presence of 2, 5 and 10 attackers respectively reduce the expected profitability for each attacker by 51.0%, 81.4% and 91.5% to 0.45, 0.17, 0.08 ETH (67, 25, 12 USD), given a victim that transacts 20 ETH to DAI on Uniswap with a transaction pending on the P2P layer for 10 seconds before being mined. If the blockchain is congested (i.e. the victim transaction remains pending for longer than the average block interval), we show that the break-even of the attacker becomes harder to attain.

- DEX security vs. scalability tradeoff. Our work uncovers an inherent tension between the security and scalability of an AMM DEX. If the DEX is used securely (i.e. under a low or zero price slippage), trades are likely to fail under high transaction volume; and an adversarial trader may profit otherwise.

II. DECENTRALIZED EXCHANGES

At the root of decentralized exchanges are blockchains. Blockchains, such as Bitcoin [48], enable peers to transact without trusting third-party intermediaries. The core component of a blockchain is a hash-linked chain of blocks [15], where miners form blocks as a data-structure which accumulates transactions. Blockchains which allow the execution of smart contracts [53], constitutes the basic building block for exchanges. A crucial aspect of this paper is that in most blockchain designs, transactions are executed in the sequence in which they are written into a block. This sequence dependence matters for blockchain-based exchanges, and will be detailed extensively in Section III. An exchange is built out of three main components: a price discovery mechanism, a trade matching engine, and a trade clearing system. Blockchains allow these components to be encoded within a smart contract to construct a decentralized, or non-custodial exchange [33], [52], [5], [43], [39], [23]. The non-custodial property guarantees that a trader retains custody over their assets at any point in time. If all exchange components are implemented within smart contracts, the exchange qualifies as an on-chain DEX. If only the trade clearing is realized within a smart contract, the exchange may be centralized but can retain the non-custodial property [39].

A. DEX components

A DEX is a game between a liquidity provider and taker.

- Liquidity Provider: a market participant that provides liquidity (financial asset trade offers).
- Liquidity Taker: a market participant that buys or sells one asset in exchange for another asset, by taking the liquidity offered by a liquidity provider.

Further, we distinguish between two varieties of DEX, depending on their mechanisms of price discovery.

- Order Book: a list of buy and sell orders for a particular asset, where each order stipulates a price and quantity. A liquidity provider quotes bid and ask prices, with an associated volume, competing for liquidity taker order flow [5], [43], [39], such that a match between supply (from a liquidity provider) and demand (from a liquidity taker) is facilitated (also referred to as market making).

- Automated Market Maker (AMM): A predefined pricing algorithm automatically performs price discovery and market making, using assets within liquidity pools [52], [4]. Liquidity providers are, therefore, not required to monitor the market to adjust bid and ask prices. Liquidity takers can directly trade against the AMM liquidity. Such automation also serves to reduce the number of on-chain transactions, making such mechanisms particularly suitable for smart contract-based DEXs given an underlying blockchain that supports only a limited number of transactions per second (tps).

B. AMM Mechanism

We denote with X/Y an asset pair composed of asset X and Y. An AMM asset pair X/Y consists of two liquidity pools, respectively for each asset:

- Asset X liquidity pool (x ∈ R\(^+\)): Total amount of asset X deposited by liquidity providers.
- Asset Y liquidity pool (y ∈ R\(^+\)): Total amount of asset Y deposited by liquidity providers.
**Definition 1.** The state (or depth) of an AMM market $X/Y$ is defined as $(x, y)$, where $x$ is the amount of asset $X$, and $y$ is the amount of asset $Y$ in the liquidity pool. The state at a given blockchain block $N$ is denoted $(x_N, y_N)$.

AMM DEXs support the following actions.

**AddLiquidity:** A liquidity provider deposits $\delta_x$ of asset $X$, and $\delta_y$ of asset $Y$ into the corresponding liquidity pools (cf. Equation 1).

$$ (x, y) \xrightarrow{\text{AddLiquidity}(\delta_x, \delta_y)} (x + \delta_x, y + \delta_y) $$ (1)

**RemoveLiquidity:** A liquidity provider withdraws $\delta_x$ of asset $X$, and $\delta_y$ of asset $Y$ from the corresponding liquidity pools (cf. Equation 2).

$$ (x, y) \xrightarrow{\text{RemoveLiquidity}(\delta_x, \delta_y)} (x - \delta_x, y - \delta_y) $$ (2)

**TransactXforY:** A liquidity taker can trade $\delta_x$ of asset $X$, increasing the available liquidity of asset $X$, in exchange for $\delta_y = f(\delta_x - c_x(\cdot)) - c_y(\cdot)$ of asset $Y$, decreasing the available liquidity of asset $Y$ (cf. Equation 3). $c_x(\cdot), c_y(\cdot)$ represent the trade fees in asset $X$ and $Y$ respectively. $f(\cdot)$ calculates the amount of asset $Y$ purchased by the liquidity taker. Each AMM exchange may chose a custom pricing function $f(\cdot)$ for governing the asset exchange [1]. Note that the exchange asset pricing cannot be determined by the exchange asset price $P$. This state change may induce unexpected slippage resulting in an execution price $P \neq E[P]$.

**Expected Execution Price ($E[P]$):** When a liquidity taker issues a trade on $X/Y$, the taker wishes to execute the trade with the expected execution price $E[P]$ (based on the AMM algorithm and $X/Y$ state), given the expected slippage.

**Execution Price ($P$):** During the time difference between a liquidity taker issuing a transaction, and the transaction being executed (e.g. mined in a block), the state of the AMM market $X/Y$ may change. This state change may induce unexpected slippage resulting in an execution price $P \neq E[P]$.

**Unexpected Price Slippage ($P - E[P]$):** is the unexpected slippage over the expected price.

**a) Slippage Example:** For example, a liquidity taker, who intends to trade 1 asset $X$ for 20 asset $Y$ at an exchange, results in a price of 0.05, quoted in units of asset $Y$. However, by the time the AMM DEX executes this transaction, if the price increases to 0.1, the liquidity taker would only receive 10 $Y$ for 1 $X$. The unexpected slippage, in this case, is $0.1 - 0.05 = 0.05$. Slippage can also be negative, i.e. a liquidity taker can receive more asset $Y$ than expected. If the execution price above decreases to 0.25, the liquidity taker would receive 40 $Y$ for 1 $X$, with a corresponding unexpected slippage of $0.1 - 0.25 = -0.15$.

**III. Sandwich Attacks on AMM DEXs**

In traditional financial markets, the predatory trading strategy of front-running involves exploiting (typically non-public) information about a pending trade, expected to materially change the price of an asset, by buying or selling the asset beforehand [6]. If the asset is expected to rise (fall) in price as a result of the trade, the front-

| AMM State 1 | AMM State 2 |
|-------------|-------------|
| Liquidity $X$ ($x$) | 100 | 1,000 |
| Liquidity $Y$ ($y$) | 10 | 100 |
| Product ($k = xy$) | 1,000 | 100,000 |
| Purchase amount $X$ | 1 | 10 |
| AMM Price $Y/X$ | 0.1000 | 0.1000 |
| $E[P]$ $Y/X$ | 0.1010 | 0.1111 |
| Expected slippage | 0.0010 | 0.0011 |
| Slippage rate | 1.01% | 11.11% |

**TABLE I:** Example price slippages on an AMM DEX.
runner will seek to buy (sell) the asset before the large pending transaction executes. AMM DEXs aim to mitigate malpractice by providing complete transparency about the available liquidity for assets $X$, $Y$, all pending and performed trades, and therefore removing the role played by non-public information. However, AMM DEXs also exacerbate malpractices by quoting asset prices in a fully deterministic way, providing relative certainty over the expected price impact of a trade. This enables a front-running adversary to perform attacks with predictable outcomes. In the following, we study two sandwich attacks on constant product AMM asset exchanges:

1) Liquidity taker attacks liquidity taker.
2) Liquidity provider attacks liquidity taker.

In each case, the fundamental intuition is that the delay in the time taken for a transaction to execute allows an adversary to profit by exploiting the knowledge of the direction of a price change. The attacks are called sandwich attacks because a victim transaction is \textit{sandwiched} between adversarial transactions.

A. System Model

We consider a blockchain P2P network, where a victim initiates trades on an AMM DEX (cf. Figure 2). An adversary observes pending victim transactions (i.e. not yet mined transactions within the memory-, or mempool) through a spy node (e.g. a custom Ethereum client), and a miner chooses to include transactions within a block according to a policy (cf. Section VI). A victim transaction trades a crypto-currency asset (such as ETH, DAI, SAI, VERI) to another crypto-asset. We do not consider blockchain forks. While blockchains typically provide delayed finality after $k$ blocks [27], [29], we consider a transaction final once included within a block.

B. Threat Model

We consider one computationally bounded and economically rational adversary $A$ (cf. Section VII for an extended threat model with multiple adversaries), that observes a zero-confirmation transaction $T_V$ from a victim trader $V$ on a blockchain P2P network. The adversarial trader can issue its own transaction $T_{A,f}$ with a transaction fee $f$. Depending on $f$, and the age of propagation, $T_{A,f}$ may be included within the blockchain prior or past $T_V$ (cf. Section VI). In this work we focus on these novel cases where the adversary is not colluding with a miner, i.e. we weaken the adversary to quantify a lower bound on the feasibility and profitability of the proposed attacks. Outside of this work, a (stronger) adversary may collude, or bribe a miner [40], [46], [14], to influence the transactions ordering within a block, or even to fork the chain as in to discard unsuccessful attacks. We moreover assume that an attack against one victim transaction is independent from other concurrent attacks towards other victim transactions.

C. Liquidity Taker Attacks Taker

In our first attack, a liquidity taker targets a victim liquidity taker who has emits on the blockchain P2P network an AMM DEX transaction (\texttt{TransactXforY}), formalizing [4]. The adversary then emits two transactions (one front- and one back-running) to exploit the victim transaction $T_V$ (cf. Figure 3). These three transactions are then unconfirmed in the blockchain P2P network, until a miner chooses to include and execute them within a block. The adversary can influence the position of the adversarial transactions, relative to the victim transaction, by paying a higher, or lower transaction fee amount (cf. Section VI).

We refer the interested reader to the Appendix A-A for the technical details of the involved transactions.
The liquidity provider D. Liquidity Provider Attacks Taker (total trade value to open a margin position. For example, profits (or losses). A trader commits a percentage of the is the process of using borrowed funds to amplify trading

We present a novel sandwich attack where a liquidity provider targets a victim liquidity taker transaction (TransactYForX) on the blockchain P2P network. Upon observing the victim transaction, the adversary emits three transactions (cf. Figure 4):

1) RemoveLiquidity (increases victim’s slippage)
2) AddLiquidity (restores pool liquidity)
3) TransactYForX (restores asset balance of X)

The (i) front-running RemoveLiquidity transaction reduces the market liquidity of the AMM DEX and increases the victim’s unexpected slippage. The (ii) back-running AddLiquidity transaction restores the percentage of liquidity A holds before the attack. Finally, (iii) the back-running transaction TransactYForX (restores asset balance of X) The (i) front-running RemoveLiquidity transaction reduces the market liquidity of the AMM DEX and increases the victim’s unexpected slippage. The (ii) back-running AddLiquidity transaction restores the percentage of liquidity A holds before the attack. Finally, (iii) the back-running transaction TransactYForX (restores asset balance of X)

E. Model Limitations

1) Margin and Leveraged Trading: Margin trading is the process of using borrowed funds to amplify trading profits (or losses). A trader commits a percentage of the total trade value to open a margin position. For example,

Fig. 3: An adversarial liquidity taker A attacks a victim taker V on an AMM DEX. Transaction TV specifies its slippage protection based on the AMM state of block N. The adversary’s goal is to include TA1, TV and TA2 in the same block N + k, k ∈ Z+ in that sequence.

D. Liquidity Provider Attacks Taker

We present a novel sandwich attack where a liquidity provider targets a victim liquidity taker transaction (TransactYForX) on the blockchain P2P network. Upon observing the victim transaction, the adversary emits three transactions (cf. Figure 4):

1) RemoveLiquidity (increases victim’s slippage)
2) AddLiquidity (restores pool liquidity)
3) TransactYForX (restores asset balance of X)

The (i) front-running RemoveLiquidity transaction reduces the market liquidity of the AMM DEX and increases the victim’s unexpected slippage. The (ii) back-running AddLiquidity transaction restores the percentage of liquidity A holds before the attack. Finally, (iii) the back-running transaction TransactYForX (restores asset balance of X)

Fig. 4: An adversarial liquidity provider A attacks a victim taker V. TV transacts asset Y for asset X.

E. Model Limitations

1) Margin and Leveraged Trading: Margin trading is the process of using borrowed funds to amplify trading profits (or losses). A trader commits a percentage of the total trade value to open a margin position. For example,

We focus only on transactions executed on Uniswap endpoints directly, not internal transactions that are routed to Uniswap.
(a) ETH/SAI market
(b) ETH/DAI market

Fig. 5: Analytical sandwich attack by a liquidity taker on a taker (Uniswap, block 9M, 0.3% fees, 0.5% unexpected slippage). If $T_V$ transacts 40 ETH for SAI, $A$ gets a max. revenue by front-running $T_V$ with a trade 18.59 ETH for 2,754.32 SAI, and back-running with 2,754.32 SAI for 18.68 ETH. This results in a profit of 0.08 ETH (11.74 USD), if $A$ bears 0.01 ETH tx fees. Note that the two sub-legends of each figure apply to both sub-figure 5(a) and 5(b).

(TA1 and TA2) of 0.01 ETH (1.97 USD) and 0.001 ETH (0.2 USD) worth of transaction fees. We observe that the greater the amount of ETH transacted by the victim, the greater is the adversarial revenue. For example, given an unexpected slippage protection of 0.5%, an adversarial taker gains a revenue of 0.01 ETH (2.03 USD) for a victim transaction trading 25 ETH to SAI on Uniswap. In contrast, the adversary gains a revenue of 0.14 ETH (20.71 USD), if the victim trades 50 ETH instead of 25.

a) Optimal Adversarial Revenue: Out of the over 1,300 Uniswap exchange markets (i.e. coin pairs to trade) an adversary may need to focus and hold liquidity in selected markets. In Figure 6 we quantify the maximum revenue an adversary can expect in a given market, conditional on a suitable victim transaction. Note that MKR has the highest liquidity (9,759.83 ETH and 2,830.27 MKR), followed by SAI (7,377.53 ETH and 1,099,040.91 SAI), WETH (5,642.08 ETH and 5,632.25 WETH), SNX (5,282.53 ETH and 572,512.14 SNX) and DAI (4,660.75 ETH and 693,706.47 DAI).

b) Minimum Profitable Victim Input: Not every victim transaction yields a profitable attack. For each of the five exchanges in Figure 6, we quantify a minimum profitable victim input \(\text{min}\) input (under 0.01 ETH transaction fee and 0.3% commission), under which an adversary will be unable to make a profit (e.g. 24.26 ETH for SAI per Figure 6). This minimum profitable victim input amount increases with the liquidity pool size (cf. Figure 7). The adversary’s optimal input increases only slightly (cf. the near horizontal line on Figure 6) with the victim transaction size, because the ETH value transacted by the victim is relatively small compared to the total amount of ETH in the Uniswap exchange. Given a fixed total slippage, we observe that markets with higher liquidity (e.g. SAI, MKR) yield higher potential revenues than lower-liquidity markets (e.g. SNX and DAI) (given the appropriate victim transaction).

B. Adversarial Liquidity Provider

Figure 8(a) and 8(b) show the revenue of an adversarial liquidity provider (cf. Section III-D), after $T_V$. We visualize the same adversarial break even lines at 0.01 ETH (1.97 USD) and 0.001 ETH (0.2 USD). Note that the adversary can only withdraw a limited amount of liquidity without triggering the slippage protection on victim’s transaction. By removing liquidity from an AMM market, the liquidity provider is forgoing a market commission (0.3% for Uniswap). To gauge profitability, we consider the following example where $T_V$ purchases 100 ETH from the SAI Uniswap exchange with 0.5% unexpected slippage. The optimal strategy is to withdraw 26.58% of the total liquidity pool, which leads to a revenue of up to 0.28 ETH (41.71 USD) for the adversary. A passive liquidity provider with 26.58% of the liquidity pool would only earn 0.08 ETH (11.91 USD) given a commission of 0.3%.

a) Who Loses Money?: Both $V$ and other honest liquidity providers lose money. $V$ purchases 100 ETH with 15,223.02 SAI without triggering a 0.5% slippage protection as a result of $A$’s front-running transaction.
Fig. 6: Sandwich attack optimal revenue for an adversarial taker when $V$ trades on five Uniswap markets (0.3% fee, $A$ breaks even at 0.01 ETH). With no adversary, $V$ only needs 15,147.28 SAI for the same amount of ETH, which is 75.74 SAI less. In addition, this $V$ transaction should increase the liquidity pool from (7,377.53 ETH / 1,099,040.91 SAI) to (7,277.53 ETH / 1,114,263.92 SAI). Post attack, the liquidity pool remains with 7,277.25 ETH and 1,114,263.92 SAI, i.e $A$ gains 7,277.53 − 7,277.25 = 0.28 ETH from the liquidity pool.

b) Optimal Adversarial Revenue: We quantify the optimal adversarial revenue in Figure 13, after subtracting the foregone opportunity cost (e.g. 0.3% for liquidity provider on Uniswap), conditional on a suitable victim transaction. We observe that the foregone commission fee is relatively stable, given a fixed total slippage, because the adversary must satisfy the victim’s slippage limit. We also quantify in Figure 9 the minimum victim input.

c) Impact of Coin Decimals: The number of decimal places for ERC20 tokens is configurable, though most of the coins have 18 decimal places. For example, USDC, which is the 7th largest Uniswap exchange at block 9M, has 6 decimal places behind the comma. In Figure 9, we plot the $k = xy$ curve for SAI if it had 17 decimal places instead of 18. The minimum victim transaction amount for $A$ increases from 43.93 ETH to 44.54 ETH, if the victim purchases ETH using SAI. The minimum victim transaction amount also increases from 45.3 ETH to 56.3 ETH for ETH to SAI transactions.

C. Overall success of the attacks

Overall, when analytically evaluated, both an adversarial liquidity taker and provider can profit by undertaking a sandwich attack. The optimal adversarial revenue, however, depends on the slippage protection setting. By fixing the unexpected slippage, the adversary’s revenue increases linearly against the amount of ETH transacted for both adversarial takers and providers. Alternatively, fixing the total slippage (unexpected + expected slippage) would yield an upper bound for both the victim transaction size and adversarial optimal profit.

V. EMPIRICAL EVALUATION

Our experimental setup corresponds to the system model in Figure 2, with a modified adversarial Parity client. We increase the maximum number of transactions in the pool of unconfirmed transactions (mempool) from the default 1024 to 2048. We design a Python script that subscribes to the modified pub/sub functionality of Parity and listens for new pending transactions of the target Uniswap market. Our script computes the profitability of any given victim transaction, and if an adversarial strategy proves profitable, the script generates and propagates the corresponding front- and back-running transaction.

We conduct both experiments (cf. Section V-A and V-B) on the main Ethereum network against the ETH/VERI
Fig. 8: Sandwich attack by a liquidity provider on a taker (Uniswap, block 9M, 0.3% fees). If \( T_V \) trades SAI for 60 ETH with an unexpected slippage of 0.5%, \( A \) can achieve a max. revenue by front-running \( T_V \) with removing 37.76% of liquidity (eq. 2.785.97 ETH and 415,030.47 SAI), and regain 37.76% of liquidity (deposit 2,749.57 ETH and 420,542.21 SAI) by back-running \( T_V \). Upon rebalancing to ETH, \( A \) gains a profit of 0.07 ETH (10.55 USD, break-even 0.01 ETH).

Fig. 9: Minimum profitable victim input on SAI Uniswap market (0.3% fee, 0.5% unexpected slippage, adversary break even at 0.01 ETH tx fees). \( A \) cannot gain any profit, if \( T_V \) trades SAI for less than 43.93 ETH. If SAI had 17 decimal places after the comma instead of 18, the min. victim transaction amount increases to 44.54 ETH.

Uniswap market and only attack our transactions. The ETH/VERI market offers the smallest liquidity (0.01 ETH and 0.07 VERI, total 3.50 USD) out of the 78 Uniswap exchanges on the Uniswap UI as of block 9M. To ensure that our results are sufficiently representative, we consider a time window of 158 days, i.e. several months. Our adversarial node runs on AWS in Ireland, (4 vCPU, AMD EPYC 7000, 2.5 GHz, NVMe SSD, max. 10 Gbps network). The experiments result in three outcomes: (i) success (all adversarial transactions are included in the same block as \( T_V \)), (ii) the front- and back-running transactions are successful, but not all adversarial transactions are included in the same block as \( T_V \), and (iii) front- or/and back-running failed.

a) Computing the adversarial transactions: Three steps allow us to compute the optimal adversarial input amount. First, the maximum amount \( A \) can transact without breaking \( V \)'s slippage protection (denoted by \( \hat{o} \)) is computed using a binary search. Second, we calculate if the attack is profitable if \( A \) inputs \( \hat{o} \). As Figure 3 and 4 suggest, if an attack is not profitable at \( \hat{o} \), then it is not profitable for any \( o < \hat{o} \). Finally, because Uniswap uses integer divisions, there might exist \( o < \hat{o} \), which results in the same or more profit. We perform a ternary search to find the optimal input.

A. Liquidity Taker Attacking Taker

We issue and attack \( T_V \) purchasing VERI with 0.001 ETH. \( T_V \) is triggered through the Uniswap UI (default 0.5% unexpected slippage) and at the time of writing default Metamask gas price (5 GWei\(^9\)). We repeat this attack 20 times, and report the results in Table II. On average, the adversary discovers \( T_V \) within 450ms, and requires less than 200ms to compute and send out \( T_{A1} \) and \( T_{A2} \). During our experiment, \( T_V \) remains in the adversary’s

\(^9\)1 GWei = \( 1 \times 10^{-9} \) ETH
of our experiment results. Compared to Section V-A, we
repeat this attack 20 times. Table II shows a summary
(2 GWei), and attack with our adversarial node. We also
indicate network congestion.

B. Liquidity Provider Attacking Taker

We initialize the adversary by adding liquidity to the
ETH/VERI Uniswap contract. We again issue $T_V$ purchasing
VERI with 0 ETH via the Uniswap UI, Metamask
(2 GWei), and attack with our adversarial node. We also
repeat this attack 20 times. Table II shows a summary
of our experiment results. Compared to Section V-A, $T_V$
remains, on average less than 10 seconds in the mempool,
which may indicate that the blockchain network is less
congested at the time of the experiment. We also observe
that the adversarial transactions are relatively closer to
$T_V$ within the block. We achieve a success rate of 20 out
of 20 attempts.

| Adversarial Taker | Adversarial Provider |
|-------------------|----------------------|
| $T_V$ Broadcast Duration | Mean: 0.45s, STD: 0.27 |
| $T_A$ Duration In Mempool | Mean: 35.88s, STD: 33.31 |
| $T_{id}$ Duration In Mempool | Mean: 48.87s, STD: 51.25 |
| A1 Block Relative Position | Mean: 0.05, STD: 0.22 |
| A1 Index Relative Position | Mean: -10.42, STD: 10.26 |
| A2 Block Relative Position | Mean: 0.70, STD: 0.92 |
| A3 Block Relative Position | Mean: 5.45, STD: 6.47 |
| A3 Index Relative Position | Mean: N/A, STD: N/A |

TABLE II: Results for the liquidity taker/provider attacks
taker. The victim’s transaction $T_V$ was triggered manually
using Metamask through Uniswap UI. Adversarial node
and victim have a clock difference of 8.781 ms ± 6.189 ms.

1) Foregone Adversarial Revenues: To understand
the financial potential of our attacks, we estimated the
theoretical revenue for the 79 exchanges of the Uniswap UI
between block 8M and 9M (i.e. recent blocks at the time of
writing), assuming a break-even at 0.01 ETH transaction
fees. Our results (cf. Table III) suggest that within the
reported 158 days, an adversary could have achieved a
revenue of 440,749.02 USD when attacking as a taker,
and 98,666.15 USD when attacking as a liquidity provider.
7.4% of transactions are profitable when attacking as a
taker, while 4.2% when attacking as a provider.

TABLE III: Estimated adversarial revenue for the 79
exchanges on the Uniswap UI, assuming an adversarial
break even cost of 0.01 ETH. Data of 158 days considered
(block 8M to 9M).

C. Slippage

To help an adversary understand how takers configure
their slippage, we plot the estimated distribution of expected slippage and maximum allowed unexpected
slippage in Figure 10. Note that we are using block $N$’s
state to calculate the slippages of a transaction mined at
block $N+1$. Therefore, these slippages are only estimates,
as we do not know the exact block state used by the taker
to create transactions. Past Uniswap transactions have
an average expected slippage of 0.58%, and an average
unexpected slippage of 1.16%.

Fig. 10: Estimated expected and maximum permitted
unexpected slippage on Uniswap transactions (block 8M
to 9M). Most takers trade with c. 1% of maximum unexpected
slippage (the Uniswap default at the time).

D. Overall success of the attacks

Our empirical results suggest that both an adversarial
liquidity taker and provider can again profit by undertaking
a sandwich attack, where the victim trades with the
Uniswap default slippage protection strategy at the time of writing this paper (0.5% total slippage). We crawled the previous transactions on Uniswap, where it shows that the most common unexpected slippage configuration is 1%, which is higher than the 0.5% default total slippage and therefore leads to higher front-running profit. Our experiments result in a high success rate (only 1 out of 40 failed), mainly because the Ethereum network was not congested, and the VERI market relatively inactive.

VI. HOW MINERS ORDER TRANSACTIONS

One crucial aspect of the potential profitability of the sandwich attacks centers on how miners order transactions within blocks. Blockchains typically prescribe specific rules for consensus, but there are only loose requirements for miners on how to order transactions within a block. To gain insight into this, we crawled the Ethereum blockchain from block 6,627,917 (where Uniswap was launched) to block 9M, constituting a total of 2,372,084 blocks, or equivalently 388 days of data. For each block, we classified the order of its transactions into one of four classes:

Empty: A block without transactions.
Gas Price: All transactions are sorted in descending order according to the gas price of each transaction.
Parity Default: Transactions are split into groups according to Parity’s prioritization (e.g., local transactions first, penalized transactions last). Then, within each group, the transactions are sorted in descending order according to each transaction’s gas price.
Unknown: Transactions are not ordered by the gas price and do not follow parity’s default strategy.

The treatment of transactions depends on the Ethereum client. At the time of writing, 78.3% of the Ethereum clients operate Geth, respectively 20.2% Parity10. Geth first sorts and separates the list of transactions into lists of individual sender accounts and sorts them by nonce. Afterward, they are merged back together and sorted by gas price, always comparing only the first transaction from each account. Parity, by default, prioritizes local and retracted transactions first, and polishes transactions with heavy computation, and then sorts by gas price. A transaction is considered local if it is received via the RPC interface, or the sender of the transaction is part of the list of locally managed accounts. Miners may choose to modify the transaction inclusion policy arbitrarily. To individually categorize each block, we first analyzed the gas price of each transaction and started by extracting the gas price only for the first transaction of each sender, while ignoring the other transactions from the same sender. We only consider the first transaction because a higher gas price transaction can be placed behind another transaction with a smaller nonce. If the extracted gas price list is sorted, we classify the block as following the “gas price” strategy. Alternatively, we verify if the gas price list consists of multiple sublists of gas prices, where the gas prices within each sublist are sorted in descending order. Each sublist represents a priority group, where transactions within the same priority group are sorted by gas price. If there are no more than four sublists of gas prices (local, retracted, normal, penalized), we classify the block as a “Parity default” block, otherwise, as “unknown”.

Because both Parity and Geth sort transactions by default by gas price, it is difficult to identify which client a miner uses. Moreover, our heuristics may misclassify blocks as being ordered using gas price instead of Parity’s default strategy. A miner could have no local transactions for the block, and all transactions are thus ordered by gas price. The number of blocks classified as Parity default should, therefore, be regarded as a lower bound. We leave it for future work to develop a more precise client fingerprinting strategy.

A. Miner Transaction Ordering Results

Our results (cf. Table V and Table IV suggest that roughly 79% of the miners order transactions based on the gas price, thus likely following Geth’s strategy, and 16% order their transaction after Parity’s default strategy. These results are consistent with the client usage distribution. We find that 2% of the blocks are empty. 3% of the miners follow an unknown ordering method. We can conclude that to position a transaction before or after a target transaction, it is, with a probability of 79%, sufficient to pay a higher (+1 Wei) or lower gas price (−1 Wei) — assuming the absence of other front-running adversaries. If two transactions pay the same gas price, according to their source code, Parity and Geth include the transactions after the FIFO principle.

Overall, we find that most miners (c. 79%) order transactions based on the gas price. The dominance of this transaction order aids an adversary because it makes it more likely for an adversary that they can control the placement of their transactions in a block by tweaking the gas price they offer for each transaction.

B. Gas Price Distribution

We investigate the gas prices for each transaction over 189,951,899 transactions included in blocks 6,627,917 until block 9M (388 days of data). A gas price of 0 ETH might indicate that the transaction belonged to the miner. We find that a transaction has an average gas price of 17.2 ± 10520.1 GWei. The median gas price is 10 Gwei and the most frequently used gas price is 20 Gwei with 23,759,990 transactions (12.5%).

VII. MULTIPLE ADVERSARIES

Our prior analysis only considers the case of a single adversary. In this section, we analyze the possible implications of multiple attackers on the feasibility and profitability of sandwich attacks through simulations.

10https://www.ethernodes.org/
Number of Attackers: Intuitively, the more adversaries we identify the following key parameters that impact the internet connection, latency, and computational power.

A. Extended Threat Model

We extend our threat model from Section III, to account for additional attackers. For simplicity, we assume that all adversaries have access to the same financial resources, internet connection, latency, and computational power.

We identify the following key parameters that impact the outcome of the simulated game:

Number of Attackers: Intuitively, the more adversaries are attacking a victim transaction $T_V$, the harder this endeavor becomes for each attacker. In the following, we consider 2, 5, and 10 adversaries, which simultaneously attack a $T_V$.

Attacker Strategy: Previous work [21] suggests two transaction fee bidding strategies, namely an adaptive (reactive counter-bidding) and a non-adaptive (blind raising) adversarial strategy. We assume that all adversaries follow the reactive counter-bidding strategy, i.e. an adversary emits a higher transaction fee bid once the adversary observes a competing transaction. This strategy is not necessarily optimal, but it may estimate a lower bound for the sandwich attack’s front-running transaction cost.

We assume all adversaries are rational and attack with the parameters determined by the strategy from Section V, i.e. each adversary attempts to maximize its profit by fully exploiting the victim transaction’s allowed slippage.

For a two-player game, we show in Table VI, the possible transaction permutations after the adversarial transactions are mined. We show that the adversary who manages to execute the first front-running transaction successfully “wins” the sandwich attack. This is because the victim transaction fails if both $A$ and $O$ execute the sandwich attack. If the other adversary is irrational and insists to execute the attack (by e.g. disregarding slippage protections), both adversaries lose, because both adversarial front-running transactions fail (e.g. $T_{A1} \checkmark$, $T_{O1} \checkmark$, $T_V \xmark$, $T_{O2} \checkmark$, $T_{A2} \checkmark$).

B. Extended System Model

a) Network Layer: The speed at which an adversarial transaction propagates within the blockchain P2P network influences the number of reactive counter-bids it receives and the time the transaction is mined. Related works have extensively studied the asynchronous nature of blockchain P2P propagation [24], [22], [29], [28]. The propagation is affected by several factors, such as the network topology, number of nodes, internet latency, bandwidth, and network congestion, etc. In our work, we assume that the adversary directly establishes a point-to-point connection with the miner and the victim. Our study thereby abstracts away the number of nodes in the network, the network topology, intermediate devices (replay nodes, routes, and switches) and TCP congestion control. Equation 5 shows how we approximate transaction propagation duration.

$$\text{Propagation Duration} = \frac{\text{Transaction Size}}{\text{Bandwidth}} + \text{Latency} \quad (5)$$

To determine the distribution of transaction sizes, we crawl raw transactions sent to the Uniswap DAI market over 100,000 consecutive blocks, starting from block 9M. Our measurements suggest a mean transaction size of 426.27 $\pm$ 68.94 Bytes, which we use as parameters for an assumed normal distribution of the adversarial and victims’ transaction sizes. For the latency and bandwidth distribution, we take the mean percentile statistics [37], [28] and apply linear interpolation to estimate the underlying cumulative probability distribution (cf. Table VII).
TABLE VI: Adversarial payoff for a two player game. Under the assumption that both players A and O are rational, A “wins” the game if the front-running transaction \( T_{A1} \) is placed in front of \( T_{O1} \), regardless of the position of the back-running transaction. A transaction that succeeds is denoted by \( \checkmark \), while a transaction that fails, is denoted by \( \times \).

| Transaction Execution Order (left to right) | Winner | Reward for Attacker A | Reward for Attacker O |
|-------------------------------------------|--------|-----------------------|-----------------------|
| \( T_{A1} \checkmark T_{O1} \times T_{V} \checkmark T_{O2} \times T_{A2} \checkmark A \) | Revenue - Fee(\( T_{A1} \)) - Fee(\( T_{A2} \)) | -Fee(\( T_{O1} \)) - Fee(\( T_{O2} \)) |
| \( T_{A1} \checkmark T_{O1} \times T_{V} \checkmark T_{A2} \checkmark T_{O2} \times A \) | Revenue - Fee(\( T_{A1} \)) - Fee(\( T_{A2} \)) | -Fee(\( T_{O1} \)) - Fee(\( T_{O2} \)) |
| \( T_{O1} \checkmark T_{A1} \times T_{V} \checkmark T_{A2} \checkmark T_{O2} \checkmark O \) | -Fee(\( T_{A1} \)) - Fee(\( T_{A2} \)) | Revenue - Fee(\( T_{O1} \)) - Fee(\( T_{O2} \)) |
| \( T_{O1} \checkmark T_{A1} \times T_{V} \checkmark T_{O2} \checkmark T_{A2} \checkmark O \) | -Fee(\( T_{A1} \)) - Fee(\( T_{A2} \)) | Revenue - Fee(\( T_{O1} \)) - Fee(\( T_{O2} \)) |

b) Transaction Fees: The transaction gas price, together with the degree of blockchain transaction congestion (i.e., competing transactions that seek to be mined), influences the pace at which a transaction is mined. In our simulations, we sample the gas price of the victim transaction from a normal distribution with a mean of 8.76±61.18 GWei, measured at the Uniswap DAI market from block 9M to 9.1M. We assume that the victim pays a sufficient transaction fee for its transaction to be mined in the next block. Empirical data suggests that the Ethereum average block interval time is 13.5±0.12 seconds [13]. Therefore, we sample the duration of the victim’s transaction in the mempool from a uniform distribution between 0 to 30 seconds. When the network is congested, transactions on Ethereum may stay in the mempool for longer than 30 seconds, sometimes even tens of minutes [13]. However, we avoid presenting our analysis on longer pending duration, as our simulation results (cf. Figure 11 and Figure 14) show that in the case of multiple players, the adversarial transaction fee at 30 seconds is likely to render the attack unprofitable.

c) Miner: We assume the miner order transactions with descending gas prices to maximize their revenue (cf. Section VI). Besides, the miners configure a price bump percentage of 10% to replace an existing transaction from the mempool. At the time of writing, Geth (used by 78.3% of the clients) configures price bump percentage to 10%, while Parity sets 12.5%.

C. Simulation Results

Figure 11 shows the expected profit of an adversarial liquidity taker (cf. Section IV) given 2, 5, and 10 adversaries, on the Uniswap DAI market at block 9M. The slippage of the victim transaction is fixed at 0.5%. The minimum profitable victim input is 14.75 ETH (2,197.30 USD).

We visualize the line where the expected revenue breaks even with simulated transaction costs. The transaction cost is calculated as the simulated gas price (cf. Figure 12) times the total gas consumed by the adversarial transaction. The gas consumption is randomly sampled from a normal distribution with mean at 85, 488 ± 34, 782 (taken from the Uniswap DAI market average gas consumption from block 9M to 9.1M). We observe that the break-even line grows exponentially until the victim transaction is mined (which is in line with the assumption of a 10% increase for each transaction price increase). We observe that the more adversaries, the more competitive the attack appears. For instance, our simulation suggests that the sandwich attack is not profitable after the victim transaction remained pending on the P2P network for 27.7, 20.3, 16.3 seconds, given 2, 5 and 10 adversaries respectively, when the victim V transacts 20 ETH for DAI. We refer the interested reader to Figure 14 for a visualization of our simulation when multiple providers attack a taker.

Our results suggest that having multiple attackers does, in expectation, divide the total revenue among the adversaries, minus the transaction fee overhead. Specifically, we find that the presence of 2, 5 and 10 attackers respectively reduce the expected profitability of the attack by 51.0%, 81.4% and 91.5% (given the victim transacts 20 ETH to DAI on Uniswap with a transaction pending on the P2P layer for 10 seconds before being mined). Note that if the blockchain is congested (i.e. \( T_{V} \) remains pending for more than 15 seconds), we observe that the break-even of the attacker becomes harder to attain.

VIII. RELATED WORK

Besides AMM DEX, other types of decentralised exchanges exist: limit order book based [5], [43], [39], auctions [23], trusted hardware [12], payment channel [42]. Front-running, and high frequency trading is related to the thoroughly studied problem of rushing adversaries to double-spend not yet mined blockchain transactions [35]. Strategically placed and malicious blockchain network nodes may control when and if miners receive transactions, which can affect the time at which a transaction is executed within the blockchain [44], [32], [30], [31].
The cryptographic literature captures front-running by allowing a “rushing adversary” to interact with a protocol [11]. The (financial) high-frequency trading (HFT) literature [7], [47] has also explored many trading strategies and their economic impact, such as arbitrage, news reaction strategies etc in traditional markets. Most of the traditional market strategies are also applicable to AMM and other decentralized exchanges [21], [8]. Daian et al. [21] in particular, introduced the concept of gas price auctions (PGA) among trading bots as well as the concept of miner extractable value (MEV). Previous studies [10] also suggest that HFT performance is strongly associated with latency and execution speed. Multiple forms of malpractice have been discovered on financial exchanges. Besides the traditional market manipulation techniques [34] (such as cornering, front-running, and pump-and-dump schemes), previous works [41] have also studied new techniques such as spoofing, pinging and mass misinformation, which leverage modern technologies such as social media and artificial intelligence. Such techniques may even be used to trick HFT algorithms [9]. To counterbalance this inherent trust, regulators conduct periodic and costly manual audits of banks, brokers, and exchanges to unveil potential misbehavior. This is a challenging task on DEX, given weak identities, and missing regulations.

IX. Conclusion

In this paper, we have presented two versions of a sandwich attack, made possible by the deterministic nature of an AMM DEX in combination with the time delay inherent on on-chain exchanges. While the transparency of DEXs is desirable, it can, however, put users assets at a security risk and allow both liquidity providers and liquidity takers to exploit unknowing traders through a combination of front and back-running. Fixing such front-running is not trivial because the smaller the allowed slippage set by a trader, the more likely the trade fails. Cryptography-based defenses moreover affect the usability of the AMM DEXes due to multiple rounds of interactions or trusted off-chain components (cf. Appendix C).

We show how under multiple competing adversaries, sandwich attacks may still remain profitable. Our work sheds light on a dilemma facing DEXs: if the default slippage is set too low, the DEX is not scalable (i.e. only supports few trades per block), if the default slippage is too high, adversaries can profit. We hope that this work draws attention to this unsolved issue and engenders future work on open, secure and decentralized finance.

Acknowledgments

We very much thank the anonymous reviewers and Nicolas Christin for the thorough reviews and helpful suggestions that significantly strengthened the paper.
Appendix A

Sandwich Attack Details

In what follows, we assume that a victim V sends a transaction $T_V$ to trade $\delta^V_y$ of Y for $\delta^V_x$ of X. The underlying AMM market starts with the initial state $s_0 = (x_0, y_0)$. Eq. 6 shows the transition in AMM state after applying $T_V$ without unexpected slippage.

$$s_0 \xrightarrow{\text{TransactXForY}(\delta^V_x)} (x^*, y^*), \quad \delta^V_y = y_0 - y^* \quad (6)$$

$x^* = x_0 + \delta^V_x, \quad y^* = \frac{x_0 y_0}{x_0 + \delta^V_x - c_x(s_0, \delta^V_x) + c_y(s_0, \delta^V_y)} + c_y(s_0, \delta^V_y) \quad (7)$

A. Liquidity Taker Attacks Taker

We assume that the adversary A has an initial state $(\delta^A_x, 0)$, and attacks by emitting $T_{A1}$ and $T_{A2}$.

$T_{A1}$: a front-running transaction, exchanges $\delta^A_x$ of X for $\delta^A_y$ of Y and is planned to execute before $T_V$ (e.g. by paying a higher transaction fee than $T_V$). This results in the state changes of Equation 8 and 9.

$$s_0 \xrightarrow{\text{TransactXForY}(\delta^A_x)} s_1 = (x_1, y_1), \quad \delta^A_y = y_0 - y_1 \quad (8)$$

$x_1 = x_0 + \delta^A_x, \quad y_1 = \frac{x_0 y_0}{x_1 - c_x(s_0, \delta^A_x) + c_y(s_0, \delta^A_y)} + c_y(s_0, \delta^A_y) \quad (9)$

$T_V$: a victim transaction modifies the state as per Equations 10 and 11.

$$s_1 \xrightarrow{\text{TransactXForY}(\delta^V_y)} s_2 = (x_2, y_2), \quad \delta^V_y = y_1 - y_2 \quad (10)$$

$$x_2 = x_1 + \delta^V_x, \quad y_2 = \frac{x_1 y_1}{x_2 - c_x(s_1, \delta^V_x)} + c_y(s_1, \delta^V_y) \quad (11)$$

$T_{A2}$: a back-running transaction exchanges $\delta^A_y = \delta^A_y - c_y(s_2, \delta^A_y)$ for $\delta^A_x$ of X and is planned to execute after $T_V$ (e.g. by paying a lower transaction fee than $T_V$). $T_{A2}$ effectively closes the adversary’s position that was opened by $T_{A1}$ (cf. Equation 12 and 13).

$$s_2 \xrightarrow{\text{TransactXForY}(\delta^A_y)} s_3 = (x_3, y_3), \quad \delta^A_x = x_2 - x_3 \quad (12)$$

$$x_3 = \frac{x_2 y_2}{y_3 - c_y(s_2, \delta^A_y)} + c_y(s_0, \delta^A_y), \quad y_3 = y_2 + \delta^A_y \quad (13)$$

Transaction $T_{A2}$ swaps the asset Y from transaction $T_{A1}$ in exchange for asset X. The corresponding profit is determined by comparing the input from $T_{A1}$ and the output $T_{A2}$ of asset X (cf. Equation 14).

$$\text{profit} = \delta^A_x - \delta^A_y \quad (14)$$

After the attack, the state of adversary becomes $(\delta^A_x, 0)$. Thus, this profit exceeds the costs (e.g. transaction fees and equipment costs), a rational adversary would undertake the attack.

B. Liquidity Provider Attacks Taker

We use $(x_N, y_N)$ to denote the state of an AMM exchange at block $N \in \mathbb{Z}$, where $x_N, y_N$ are the amounts of asset X and Y in the liquidity pool (cf. Definition 1). The liquidity provider owns a share of $L \in [0,1]$ liquidity if it deposits $\delta_{x,N} = x_N \frac{L}{\sqrt{x_N}}$ of asset X and $\delta_{y,N} = y_N \frac{L}{\sqrt{y_N}}$ of asset Y into the liquidity pool. We use $Z_N(x, y)$ to denote the USD value of assets at block $N$, where $x$ is the amount of asset X and $y$ is the amount of asset Y. Given two blockchain blocks $m, n$, where the respective AMM states differ, the deposit of $z_m = Z_m(x_m \frac{L}{\sqrt{x_m}}, y_m \frac{L}{\sqrt{y_m}})$ at block $m$, or $z_n = Z_n(x_n \frac{L}{\sqrt{x_n}}, y_n \frac{L}{\sqrt{y_n}})$ at block $n$ both results in a share of L liquidity in the respective market X/Y. For the following section, we set $z_m > z_n$.

1) Attack Profitability: We proceed by defining profitability for an adversary. $L_x, L_y, L^*_x, L^*_y \in [0, 1]$ denote the proportion of assets X and Y held by the adversary in the liquidity pools before and after the attack. Analogously, $x^A, x^A, y^A, y^A \in \mathbb{Z}^+$ denote the amounts of asset X and Y held by the adversary before and after the attack.

$$(L_x, L_y, x^A, y^A) \xrightarrow{\text{attack}} (L^*_x, L^*_y, x^A, y^A) \quad (15)$$

The state transition (cf. Equation 15) is profitable to the adversary, if the following conditions hold:

- $L^*_x \neq L_x, \quad L^*_y \neq L_y, \quad x^A \neq x^A, \quad y^A \neq y^A,$ and
- At least one of the following holds:
  - $L^*_x > L_x$, or $L^*_y > L_y$, or $x^A > x^A$, or $y^A > y^A$

Let $x^*$ and $y^*$ denote the amounts of asset X and Y in liquidity pools after the attack. The corresponding profit
is determined by comparing adversary’s states before and after the attack (cf. Equation 16, 17, 18).

\[ \Delta x = x^{A+} - x^A + (L_x^* - L_x)x^* \]  
\[ \Delta y = y^{A+} - y^A + (L_y^* - L_y)y^* \]  
\[ \text{profit} = Z(\Delta x, \Delta y) \]  

(16) (17) (18)

2) Attack Execution: We now consider an adversarial liquidity provider that owns a share \( L_x x_0, L_y y_0 \) of the total liquidity pool of a DEX X/Y market. The victim’s transaction \( T_V \) transacts asset \( X \) for \( Y \). If the adversary does not front-run \( T_V \), the AMM state changes according to Equation 19, 20. In that case, the adversary \( A \) receives a commission fee \( c \), as stated in Equation 21, 22.

\[ s_0 \xrightarrow{\text{TransactXForY}(\delta^V)} s_1 = (x_1, y_1), \quad \delta^V = y_0 - y_1 \]  
\[ x_1 = x_0 + \delta^V, \quad y_1 = \frac{x_0 y_0}{x_1 - c_x(s_0, \delta^V)} + c_y(s_0, \delta^V) \]  
\[ c^{A,T_V}_x = L \times c_x(s_0, \delta^V), \quad c^{A,T_V}_y = L \times c_y(s_0, \delta^V) \]  
\[ (L_x, L_y, x^A, y^A) \rightarrow (L_x, L_y, x^A + c^{A,T_V}_x, y^A + c^{A,T_V}_y) \]  

(19) (20) (21) (22)

This liquidity provider attempts to gain a profit through the following order of transactions.

- **\( T_{A1} \):** a front-running transaction executed before \( T_V \) (e.g. by paying a higher transaction fee than \( T_V \)). \( T_{A1} \) withdraws \( (L_x x_0, L_y y_0) \) from the liquidity pool \( (x_0, y_0) \) and results in the state changes of Equation 23 and 24.

\[ s_0 \xrightarrow{\text{RemoveLiquidity}(L_x x_0, L_y y_0)} s_1 = (x_1, y_1) \]  
\[ x_1 = x_0 - L_x x_0, \quad y_1 = y_0 - L_y y_0 \]  

(23) (24)

- **\( T_V \):** a victim’s transaction modifies the state according to Equation 25 and 26.

\[ s_1 \xrightarrow{\text{TransactXForY}(\delta^V)} s_2 = (x_2, y_2), \quad \delta^V = y_1 - y_2 \]  
\[ x_2 = x_1 + \delta^V, \quad y_2 = \frac{x_1 y_1}{x_2 - c_x(s_1, \delta^V)} + c_y(s_1, \delta^V) \]  

(25) (26)

- **\( T_{A2} \):** a back-running transaction executes after \( T_V \) (e.g. by paying a lower transaction fee than \( T_V \)). \( T_{A2} \) adds back liquidity for the adversary to maintain the same proportion of overall liquidity \( (L_x, L_y) \), modifying the state according to Equation 27 and 28.

\[ s_2 \xrightarrow{\text{AddLiquidity}(\frac{x_2}{L_x}, \frac{L + x_2}{L_y})} s_3 = (x_3, y_3) \]  
\[ x_3 = \frac{x_2}{1 - L_x}, \quad y_3 = \frac{y_2}{1 - L_y} \]  

(27) (28)

- **\( T_{A3} \):** executed after \( T_{A2} \), \( T_{A3} \) rebalances the AMM assets by converting \( Y \) to \( X \), such that the adversary retains the same amount of asset \( X \) as before the attack (i.e. the adversary holds \( x^A + c^{A,T_V}_x \) after the attack). This rebalancing process is necessary because the amount of asset \( X \) added to liquidity pool in \( T_{A2} \) exceeds the amount withdrawn from \( T_{A1} \). \( T_{A3} \) modifies the state according to the Equations 29, 30 and 31.

\[ s_3 \xrightarrow{\text{TransactXForY}(\delta^A)} s_4 = (x_4, y_4), \quad \delta^A = x_3 - x_4 \]  
\[ x_4 = \frac{x_3 y_3}{y_4 - c_y(s_3, \delta^A)}, \quad y_4 = y_3 + \delta^A \]  
\[ \delta^A = \frac{L_x x_2}{1 - L_x} + c^{A,T_V}_x - L_x x_0 \]  

(29) (30) (31)

The overall state change of this attack is described in Equation 32. At the end of this attack, the adversary \( A \) has managed to retain \( L \) proportion of the total liquidity as a result of \( T_{A2} \), retains the same total amount of asset \( X \) (as a result of \( T_{A3} \)) but increases their holding of asset \( Y \), generating a profit. Equation 33 yields the profit.

\[ (L_x, L_y, x^A, y^A) \xrightarrow{\text{attack}} (L_x, L_y, x^A + c^{A,T_V}_x, y^A) \]  
\[ \text{profit} = y^{A+} - (y^A + c^{A,T_V}_y) \]  

(32) (33)

**APPENDIX B**

**ADDITIONAL DETAILS FOR LIQUIDITY PROVIDER**

Figure 13 quantifies the optimal adversarial revenue by a liquidity provider, given no other attacks are running.

Figure 14 shows the expected profit of an adversarial liquidity provider (cf. Section IV) given 2, 5, and 10 adversaries, on the Uniswap DAI market at block 9M. The victim transaction is fixed with a 0.5% slippage protection. The minimum profitable victim input with a single adversary is 27.8 ETH (4,141.37 USD).

**APPENDIX C**

**POSSIBLE MITIGATION**

In this section we elaborate on how users could protect their trades from front-running attacks, and we discuss possible AMM design changes to strengthen their resilience.

**A. AMM Specific Settings**

We identify two primary protection possibilities that could be adopted given the existing AMM DEX designs (such as Uniswap).

1) **Allow for Zero Slippage**: The simplest protection for a user, is to not allow its trades to pay any slippage. If an adversary were to front-run, the user’s trade execution will fail. The user would not overpay the trade due to its slippage protection, but the user still is liable to pay blockchain transaction fees. This protection solution moreover is not scalable in terms of trades per second, as it would limit the number of trades an AMM DEX can execute to 1 transaction per blockchain block.
B. Cryptography-based Defences

In the following we discuss possible cryptography-based defence techniques against sandwich attacks.

1) Multi-Party Computation (MPC): The sandwich attack is possible because the current consensus protocol used in Ethereum fails to protect the actual ordering of the transactions from adversarial manipulations (i.e., fee manipulations). Thus, it is tempting for one to design an AMM DEX that has a authorized set of “trusted” nodes to faithfully sequence actions from liquidity takers before getting the actions executed by the smart contract. This proposal, however, makes AMM DEX like Uniswap no longer permissionless, and the authorized set can again manipulate the order of the transactions. To address the later, Kelkar et al. [36] propose a new set of Byzantine Consensus protocols, that achieve a fair ordering of received transactions. Thus, one can require the authorized set of nodes to run such protocol to achieve order fairness among transactions.

2) Commit-and-Reveal Protocols: A commitment scheme is a two-round protocol that allows one to commit to chosen values (i.e., function, input) while keeping those values hidden from others (hiding) during the committing round, and later during the revealing round, s/he can decide to reveal the committed value. The commitment schemes are binding if and only if the party cannot change the value after committing to it. We briefly discuss the use of commitment scheme to prevent front-running in the following.

a) Standard Commit-and-Reveal Protocol: To prevent the sandwich attack in AMM DEX, one can use commitment schemes to sequence actions of traders during committing round and execute actions during revealing round. In particular, during committing round, traders commit to function calls (i.e., `TransactXforY`, `addLiquidity`, `removeLiquidity`) via commitment transactions, and the ordering of function calls is determined based on the order of commitments while the function calls are hidden due to the hiding property of a commitment scheme. In the revealing round, parties can decide to reveal the function calls, and the AMM DEX will execute the transactions according to the order of the commitments appeared in the committing round.

One of the limitations of commit-and-reveal protocol is its usability, as it requires participants to be aware of both rounds of the protocol to complete their actions. Another limitation is that adversary is still able to probabilistically perform sandwich attack. In particular, because the committing round is transparent, the adversary can see other traders’ commitments and commits several transactions before and after the commitments of honest traders. S/he can reveal only those transactions that are profitable.

b) Commit-and-Reveal Protocol tailored for the Ethereum Blockchain: To hide the committing phase from the adversary, Breidenbach et al. [17] proposed submarine commitments via contract creation in Ethereum (i.e.,

**Fig. 13: Optimal adversarial revenue under a sandwich attack by a liquidity provider, when V sells assets for ETH on five Uniswap exchanges (0.3% fee, adversarial break-even at 0.01 ETH).**

2) Limit Liquidity Taker’s Input: Recall that sandwich attacks are only profitable if the victim’s input amount is above a threshold, which we named it as the minimum profitable victim input (cf. Section IV). This threshold depends on both the total liquidity of the market and the AMM DEX’s design and configuration (pricing formula, fees and etc.). One possible protection is to disable transactions above the minimum profitable victim input in the smart contracts.
CREATE2 EVM opcode [2]). The essence of this approach is to hide commitment transaction among newly generated Ethereum addresses. In particular, a subarctic committee scheme contains the following phases:

**Committing Phase:** to commit in a submarine committee scheme, the liquidity taker with address \( \text{Addr}_{\text{Taker}} \) posts a transaction \( T_{\text{com}} \) that sends some fund, \( \text{val} \), to an address \( \text{Addr}_{\text{com}} \). \( \text{Addr}_{\text{com}} \) is a commitment of the form:

\[
\text{Addr}_{\text{com}} = H(\text{Addr}_{\text{AMM-DEX}}, H(\text{Addr}_{\text{Taker}}, \text{key}), \text{code})
\]

where \( H(\cdot) \) is Keccak-256, \( \text{key} \) is the transaction specific key to AMM DEX (e.g., a concatenation of action \( \text{transact.XforY} \), input \( \delta_x \), and 256-bit randomness \( r \)), and \( \text{code} \) is the EVM init code of the refund contract that can send any money received to the \( \text{Addr}_{\text{AMM-DEX}} \).

**Revealing Phase:** To reveal, \( \text{Addr}_{\text{Taker}} \) sends to \( \text{Addr}_{\text{AMM-DEX}} \), key value \( \text{key} = (\text{action}|\text{input}|r) \), the transaction data \( T_{\text{com}} \), the \( \text{commitBlock} \) (the block number includes \( T_{\text{com}} \)), and a Merkle-Partricia proof, \( \pi_{\text{com}} \), which proves the membership of \( T_{\text{com}} \) in \( \text{commitBlock} \). With \( \text{commitBlock}, \pi_{\text{com}}, \text{and } T_{\text{com}}, \text{Addr}_{\text{AMM-DEX}} \) verifies that \( T_{\text{com}} \) occurred in \( \text{commitBlock} \). And after learning key, \( \text{Addr}_{\text{AMM-DEX}} \) can recompute \( \text{Addr}_{\text{com}} \), verifies the deposit balance \( \text{val} \), and proceeds with action and input (i.e. \( \text{transact.XforY}(\delta_x) \)).

**Deposit Collection Phase:** from \( \text{code} \), \( \text{Addr}_{\text{AMM-DEX}} \) can use CREATE2 opcode to create an instance of the refund contract at \( \text{Addr}_{\text{com}} \) and collect \( \text{val} \) from \( \text{Addr}_{\text{com}} \).

3) **Confidential Transactions:** Another potential attempt to mitigate front-running attack is to hide the details of the transaction sent to AMM DEX by adapting several techniques [50], [18], [16], [26] for confidential transactions. However, as pointed out by Eskandari et al. [25], to prevent front-running, one needs to hide:

1) The name of the functions (i.e. \( \text{Transact.XforY}, \text{addLiquidity}, \text{removeLiquidity} \)) being invoked
2) The parameters supplied to the functions (i.e. \( \delta_x, \delta_y \))
3) The current state of the DEX (i.e. \( (x,y) \)).

While systems like Hawk [38] and Ekiden [19] try to achieve all three properties for arbitrary functions, they rely on off-chain components (i.e. trusted execution environment) for maintaining encrypted states and proving the correctness of state transitions. On the other hand, a proposal for Ethereum blockchain, Zether [18], tries to achieve (2, 3) for a specific type of function (i.e. money transferring). The states in Zether are the ElGamal encryptions of account’s balances. The state transitions can be made due to the correctness of the non-interactive zero-knowledge proof (NIZK) system used in their construction and the (additively) homomorphic properties of ElGamal encryption.

However, recall that in an AMM DEX, an action like \( \text{Transact.XforY}(\delta_x) \) requires the contract to send an \( \delta_y \) of asset \( Y \) back to the liquidity taker where \( \delta_y \) is computed based on the pricing function \( f(\cdot) \) and the current pool’s state \( (x,y) \). Therefore, if one decides to use a system that relies on NIZK systems like Zether for AMM DEX with hidden state, the trusted off-chain components are needed to generate cryptographic proofs from the unencrypted state to trigger the transferring back function.

Moreover, regardless of the privacy techniques used for hiding transaction details and pool’s states, we show that it is impossible to achieve the confidentiality for the third property for a constant product AMM. In particular, at the initial pool state \( s = (x,y) \), an adversary can issue two consecutive actions, \( \text{Transact.XforY}(\delta_x), \text{Transact.XforY}(\delta_y') \), and it obtains \( \delta_y \) and \( \delta_y' \) determined by the current state \( (x,y) \). The adversary can solve the system of two equations and two unknown \( (x,y) \) to determine the current pool state.