MEVisor: High-Throughput MEV Discovery in DEXs with GPU Parallelism

Weimin Chen
The Hong Kong Polytechnic University
cswchen@comp.polyu.edu.hk

Xiapu Luo
The Hong Kong Polytechnic University
csxluo@comp.polyu.edu.hk

Abstract*—Decentralized finance (DeFi) is an emerging financial service on blockchain, enabling automatic and anonymous transactions. Within DeFi, decentralized exchanges (DEXs) maintain reserves of a pair of tokens and determine the exchange rate to swap tokens. However, DEXs also create opportunities for Maximal Extractable Value (MEV), where attackers include, exclude, or reorder DEX transactions to exploit price discrepancies of tokens and extract profit. Uncovering MEV opportunities requires high throughput, as the 12-second block interval and the vast search space impose strict time constraints. However, existing tools suffer from low throughput, as they rely on CPU-bound execution, which is hindered by frequent state forking and slow DEX execution. In this paper, we take the first step in leveraging GPU parallel computing power to boost MEV-search throughput in arbitrage and sandwich strategies. More precisely, we compile an MEV bot into a GPU application and then launch thousands of GPU threads to search for profit in parallel. To this end, we design new solutions to address three major challenges: designing cheatcodes to simulate transactions on GPU, proposing a memory manager to reduce GPU memory usage, and designing strategy-aware mutations to improve input diversity. We implement a prototype named `MeVisor` that runs DEXs on GPUs and searches for MEV using a parallel genetic algorithm. Evaluated on 3,941 real MEV cases from Ethereum, `MeVisor` achieves 3.3M-5.1M transactions per second, outperforming the CPU baseline by 100,000x. In a large-scale study of Q1 2025 data, `MeVisor` estimates MEV opportunities ranging from 2 to 14 transactions, yielding at most $1.1 million in MEV profit.

I. Introduction

Decentralized finance (DeFi) is a rapidly growing ecosystem of peer-to-peer financial services built on blockchains, particularly Ethereum [62]. DeFi applications are implemented as smart contracts, which are self-executing programs running on the Ethereum Virtual Machine (EVM [62]). These contracts enable permissionless markets and anonymous transactions. As a fundamental component of DeFi, decentralized exchanges (DEXs) hold a pair of tokens and adjust their prices based on market supply and demand. Traders swap tokens on DEXs, generating $13.7 billion in daily trading volume [33] and millions of transactions [23]. During a swap, traders may incur losses due to slippage, which is the difference between the expected price and the actual execution price. Such price discrepancies create MEV opportunities for high-frequency searchers, producing more than $300K in daily profits [18].

MEV [12] is the profit extractable by including, omitting, or reordering transactions within a block. The resulting ordered set of transactions, known as a bundle, constitutes the MEV payload. A classic MEV strategy is *arbitrage*, where an attacker inserts a transaction to buy a token on one DEX and immediately sells it on another at a higher price. Another common strategy is *sandwich*: The attacker first front-runs a victim's buy trade by purchasing the token immediately beforehand to raise the price; after the victim's trade executes with slippage, the attacker back-runs by selling the token and captures the profit created by the victim's price impact.

Although MEV is sometimes described as the "invisible tax" of DeFi, it can also contribute to market by adding liquidity and attracting new capital. Several blockchain features further accelerate the growth of MEV. First, after Ethereum's 2022 Proof-of-Stake upgrade [39], attackers without mining power can still submit their MEV bundles through relays [29], [59]. Second, Ethereum synchronizes peers every 12 seconds [39], provides a uniform search window during which attackers worldwide can compete for MEV opportunities. Third, attackers with limited capital can also submit high-cost bundles using flash loans [56]. A flash loan is a DeFi service that provides an instant, collateral-free loan. It allows an attacker to borrow tokens to execute MEV transactions and then repay the loan using the extracted profit.

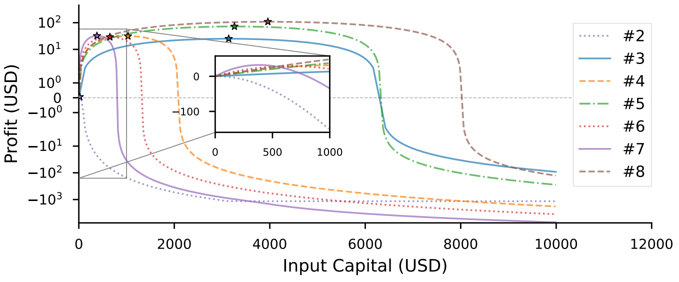
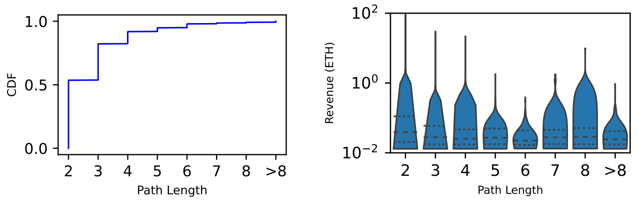


Fig. 1: Profit trends of the motivating examples listed in Table X.

Searching for MEV opportunities is fundamentally an optimization problem aimed at maximizing profit by selecting bundles within a fair but limited time window. In our threat

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240093
www.ndss-symposium.org

model, attackers can insert transactions in specific positions to front-run or back-run a victim [12] to build a bundle. While MEV strategies specify where these transactions should appear, the inserted transactions themselves are not predetermined. Transactions must be synthesized with an appropriate amount of initial capital and a corresponding swap path across DEXs. The initial capital is the input token amount used throughout the MEV path. Prior work [15] shows that the capital optimization for Uniswap V2 DEXs [2] is a convex optimization problem, and can therefore be solved both reliably and efficiently. Figure 1 illustrates the profit curves of our seven motivating examples (Table X) as functions of initial capital, demonstrating convexity even when the MEV path spans multiple DEXs to Uniswap V3 [1], and Sushiswap [57]. However, identifying an MEV path is non-trivial due to the explosion of the search space. For sandwich attacks, although the attacker typically searches only two trades, selecting the optimal pair incurs quadratic complexity. With n tradable pools, the two-leg ordered pairs scale as $O(n^2)$, and when combined with a superimposed size grid of k, the overall complexity becomes $O(kn^2)$. Searching for profitable sandwich opportunities is computationally intensive, as it must account for slippage across all pairwise DEX combinations in real time. Arbitrage attacks have even higher complexity. Identifying an MEV arbitrage path of length s across n tradable DEXs requires evaluating $\frac{n!}{(n-s)!}$ combinations. This factorial growth forces searchers to restrict exhaustive search to short paths. Using data from [61], we analyzed 30,646 historical arbitrage transactions from January to June 2025. The cumulative distribution in Figure 2a shows that 82.21% of arbitrage events involve fewer than three DEXs. Despite this trend, longer arbitrage paths remain valuable, as they often contain profit opportunities overlooked by most attackers. Figure 2b shows that long arbitrage paths yield arbitrage profits.



(a) CDF of path lengths. (b) Distribution of revenue.

Fig. 2: Empirical study on 30,646 arbitrage incidents.

The explosion of the search space significantly limits an searcher's ability to exploit MEV opportunities under real-world constraints. On Ethereum, where the blockchain is updated approximately every 12 seconds, attackers must evaluate and rank potentially millions of bundles within seconds to remain competitive. Table I summarizes the existing MEV detection tools, all of which struggle with low throughput when identifying practical MEV opportunities. DEFIPOSTER-SMT [64] was the first system to encode arbitrage MEV as symbolic constraints and solve them using SMT. However, its scalability is limited by the path explosion inherent to symbolic execution. DEFIPOSTER-ARB [64] and CFMMROUTER [15] reduce this explosion by searching for optimal swap cycles within a restricted subset of DEXs. These approaches are specific to arbitrage but cannot be easily generalized to other MEV strategies. CFF [4] improves generality by formally verifying smart contract interactions across different MEV strategies. Nonetheless, its scalability remains limited because each formal model is tailored to a specific DEX, preventing broad cross-DEX applicability. Overall, these deterministic methods remain constrained by the size of the bundle space and often fail to detect complex or previously unseen MEV paths. Lanturn is the only non-deterministic approach. It runs an infinite optimization loop that learns MEV strategies from historical transactions. In each round, it evaluates bundles on a testbed built by Hardhat [35], which must locally simulate consensus protocols. Although this design improves scalability for complex MEV opportunities, Lanturn is still limited by EVM execution overhead, including consensus simulation and the cost of repeatedly forking and executing transactions. As a result, it achieves only 121 transactions per second on 44 cores, leaving significant room for improvement. While Lanturn includes consensus simulation and frequent forking, advanced MEV searchers can often omit consensus protocol and avoid unnecessary full-node forking by incrementally retrieving blockchain states.

TABLE I: Comparison of existing MEV detectors. Scalable: Capable of handling a growing number of DEXs. Generalized: Supports MEV strategies beyond arbitrage. EVM-free: Search without relying on EVM execution.

	Type	GPU	Scalable	Generalized	EVM-free
`MeVisor`	Genetic Algorithm	✓	✓	●	✓
Lanturn [5]	Machine Learning	✗	✓	●	✗
CFF [4]	Formal Verification	✗	✗	◐	✓
CFMMROUTER [15]	Optimal Routing	✗	✓	○	✗
DEFIPOSER-ARB [64]	Shortest Path	✗	✓	○	✗
DEFIPOSER-SMT [64]	Symbolic Execution	✗	✗	○	✓

● Fully support ◐ Partially support ○ Missing

To uncover opportunities to improve baselines' throughput, we conducted a motivating study that profiles the time distribution across each stage of the their workflow (§ II-D): ① forking from the Ethereum mainnet (main network) [62] for local testing; ② constructing MEV bundles; and ③ evaluating bundles through transaction execution. The results indicate that Steps ② and ③ together account for 99% of total computation time. Execution time also grows with the length of the MEV path. This heavy dependence on smart contract execution and consensus simulation severely limits MEV search throughput. Finally, although prior work [4], [5] attempts to parallelize MEV search using multithreading or distributed computing, CPU hardware limitations continue to be a fundamental bottleneck.

In this paper, we introduce `MeVisor`, the first MEV detector which significantly improves throughput of arbitrage and sandwich searching by harnessing the parallel computing power of GPUs. More precisely, we compile the smart contracts of the MEV bot into a GPU application and implement a parallel genetic algorithm to search for profitable bundles across the GPU-compiled smart contracts. For scalability, our GPU compiler is language-agnostic and supports thousands of DEXs that follow the protocols of Uniswap V2 [2], Uniswap V3 [1], and Sushiswap [57]. Attackers can deploy generalized MEV strategies using MEV bots for concurrent search. There are four properties of MEV search that enable effective GPU acceleration and significantly increase throughput: **P1: Data parallelism.** DEXs adhering to the same protocol share execution code and differ only in smart contract states, such as token reserves and liquidity amount. GPUs execute the same instruction on different DEX data (SIMD [43]) to evaluate profit in parallel (see § II-C). **P2: Light data movement.** MEV search requires only the DEX states, minimizing host-to-GPU data movement. This reduces memory overhead and improves throughput [19]. **P3: Dense computation.** MEV searchers must evaluate millions of bundle candidates within each 12-second interval. The wide SIMD lanes of GPUs support large-scale searches within this tight window. **P4: Standalone execution.** Since smart contracts do not rely on system calls, it is feasible to execute the MEV bot entirely on the GPU, avoiding performance bottlenecks caused by host-GPU interactions.

Given an MEV strategy, we first generate a smart contract MEV bot that can estimate profit of MEV bundles (§ III-B). We then compile it to GPU code (i.e., PTX [50]), enabling concurrent evaluation of candidate bundles on the GPU and transforming the parallel MEV search into an SIMD-parallel workload (§ III-C). Across the pipeline, TRANSLATOR performs PTX compilation only a single time, prior to the MEV search phase. SEARCHER is a GPU-based genetic algorithm [14] that discovers MEV opportunities by executing thousands of threads of the MEV bot in parallel (§ III-D). Discovered solutions are validated in a mainnet fork environment to eliminate false positives. `MeVisor` avoids the overhead of EVM forking by performing forking only during final validation, after the search phase completes.

Developing `MeVisor` is non-trivial due to three key challenges: **C1:** Typical MEV attackers [5] send ordered calls to EVM to evaluate the profit of their bundles. However, we expect to discard EVM but run an MEV bot on GPU directly for a higher throughput. The MEV bot needs to simulate MEV transactions within one smart contract execution without using EVM. **C2:** Cross-contract calls are necessary for executing smart contracts on GPUs; however, instantiating separate EVM call frames incurs significant overhead due to the inefficiency of dynamic memory allocation on GPU hardware. **C3:** Bundle diversity critically influences the performance of SEARCHER. However, transactions generated through mutation may violate the constraints of specific MEV strategies, resulting in redundant evaluations and reduced bundle diversity.

To tackle **C1**, we design two primary cheatcodes for the MEV bot, namely `vm.swap` and `vm.apply` (§ IV-A), which are two GPU functions that simulate Ethereum transactions on the GPU. `vm.swap` simulates the attackers' transactions to DEXs for swapping tokens. `vm.apply` simulates the victim transaction of a sandwich attack for mock the price impact caused by the victim transaction. Together, these cheatcodes support MEV strategies such as arbitrage and sandwich. To address **C2**, we implement a GPU-specific memory manager that eliminates dynamic allocation while preserving call frame isolation (§ IV-B3). To address **C3**, we introduce a swap graph that systematically guides the mutation process, ensuring that generated transactions consistently satisfy the constraints of targeted MEV strategies (§ IV-D3).

We implemented a `MeVisor` prototype using approximately 2,500 lines of C/C++ and 2,000 lines of CUDA [48]. To evaluate its effectiveness, we compared `MeVisor` against state-of-the-art tools using a ground-truth benchmark derived from historical MEV data, including 2,380 arbitrage and 1,561 sandwich activities. The experimental results show that `MeVisor` achieves throughputs of 1.43M execs/sec and 1.72M execs/sec on the arbitrage and sandwich datasets, respectively. This represents a 100,000x improvement over the fastest CPU baseline [5] and a 5x speedup over the fastest known GPU baseline [9]. This high throughput enabled `MeVisor` to uncover $1,123,642.49 in MEV profit during Q1 2025.

Our contributions:

- We design and implement `MeVisor`, the first GPU-based MEV detector for both arbitrage and sandwich, which uncovers profit in parallel using GPU acceleration.
- We conduct a comprehensive benchmark evaluation, demonstrating that `MeVisor` detects significantly more profit in less time than existing tools, with a 100,000x higher throughput.
- We evaluate `MeVisor` on real Ethereum data from Q1 2025, where it discovers $1,123,642.49 in MEV profit (§ V-C).

II. BACKGROUND

A. Smart Contracts for DEXs

Smart contracts are written in high-level languages such as Solidity [25] or Vyper [60] and compiled into a uniform format known as EVM bytecode. The EVM interprets stack-based instructions of this bytecode by unpacking arguments from an input buffer called *calldata*, updating smart contract states via the *stack* and *memory*, storing persistent data in a key–value mapping called *storage*, and writing the final output to a return buffer named *retdata*. Each EVM instruction has an associated execution cost, measured in gas.

DEX is an application of smart contract, enabling trading of tokens [1], [2], [11], [57]. A DEX consists of a liquidity pool that maintains reserves of tokens and determines the exchange rate. DEXs adhering to the same protocol typically rely on a common router contract to consolidate complex operations, such as path optimization and multi-hop execution. In DEXs, token trading is implemented entirely as smart contract execution on the EVM. MEV attackers typically deploy a smart contract, known as an MEV bot, to execute their bundles and make profit.

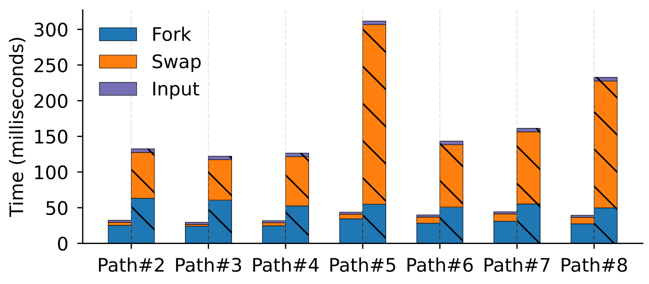


Fig. 3: Time distribution across the three MEV search steps per run. The left bars represent Foundry. The right bars represent Lanturn, used as the SOTA baseline for comparison.

B. MEV Competition

MEV refers to the maximum profit a searcher can extract by strategically including, excluding or reordering transactions within a block [12]. MEV has become a dominant transactions of high-frequency trading on Ethereum. Token price discrepancies across DEXs, often caused by DeFi trades, create MEV opportunities. Readers can refer to Appendix A for more details of MEV attacks to DEXs.

MEV opportunities are inherently transient due to rapid market adjustments and intense competition among sophisticated searchers. Since the launch of Proof-of-Stake in 2022 [39], each 12-second block slot imposes strict timing constraints on the MEV search. The upcoming enhanced PBS (ePBS) [17] further refines this mechanism by subdividing each slot into smaller phases. This evolution increases transparency and fairness but also intensifies competition.

C. GPU vs. CPU execution models.

A CPU is designed for latency-oriented serial processing, emphasizing complex control logic and branch prediction to optimize single-thread performance. Each CPU core executes sophisticated instruction pipelines, aiming to minimize the execution time of each individual task.

In contrast, a GPU follows a throughput-oriented model, optimized for massive data parallelism. It contains thousands of cores grouped into streaming multiprocessors, each executing many threads concurrently under the SIMD paradigm. Threads within a warp (typically 32) execute the same instruction in lockstep on different data, making GPUs highly efficient for uniform, data-parallel workloads such as smart contract simulations with similar control flow.

D. Motivation

MEV search is fundamentally an optimization problem: it aims to maximize profit by selecting bundles and evaluating their profit through smart contract execution. To understand the computational bottlenecks of existing tools and opportunities for acceleration, we conducted a motivating study.

Findings. We randomly selected seven historical arbitrage MEV cases with transaction lengths ranging from 2 to 8 (see Table X). We used two tools, Lanturn and Foundry, to replay MEV transactions, running each tool over 1,000 iterations. Figure 3 shows the average time distribution across their three steps of each MEV search round: ① forking from the mainnet, ② selecting an input bundle, and ③ evaluating profit via smart contract execution. Our analysis reveals that input generation takes less than 1% of the total time, while EVM execution and state forking dominate the overall latency. Consistent with prior studies [5], [9], EVM execution emerges as the primary performance bottleneck, and its impact increases with bundle length. Its impact grows with the increased length of transaction bundles. We use Foundry [32] as an additional baseline. It generates random transaction bundles and executes them on the fastest CPU-side EVM [7]. Overall, Foundry achieves an upper-bound throughput of 141.42 transactions per second, spending 80.89% of execution time on forking (27.87 ms) and 18.82% on smart contract execution (6.49 ms). In contrast, Lanturn, the state-of-the-art MEV searcher, achieves only 30.35 txs/sec because it uses a slower EVM [35] implemented in TypeScript. Its smart contract execution is 17.8x slower than Foundry's, and forking is 1.98x slower. Although Lanturn attempts to use multi-threaded CPU execution to mitigate overhead due to ③, its performance remains fundamentally bounded by hardware limitations.

Main Idea. MEV search is a compute-intensive task that requires the rapid evaluation of millions of bundles within Ethereum's 12-second block interval. Our intuitive idea is to improve throughput by executing bundles concurrently on a GPU using SIMD parallelism, thereby accelerating MEV discovery. Specifically, `MeVisor` compiles the smart contracts of both the MEV bot and the target DEXs into a unified GPU application (in PTX format), enabling parallel evaluation of thousands of input bundles. A single RTX 3090 GPU offers over 100x greater thread-level parallelism than high-end CPUs, making it well-suited for this task. Promising bundles identified on the GPU are selectively replayed on the CPU within a private fork to verify correctness. This design significantly reduces the overhead of step ③, as it requires only a single fork for the entire search process. Additionally, concurrent GPU execution mitigates the latency introduced by step ②.

III. Overview

In this section, we present a high-level overview of `MeVisor`, including its architecture and key design choices that address the three technical challenges (**C1–C3**).

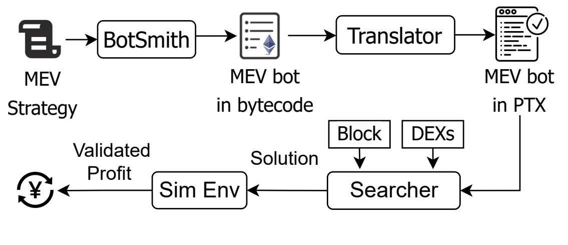


Fig. 4: Overview of `MeVisor`.

A. `MeVisor`'s Architecture

Figure 4 presents the architecture of `MeVisor`, which consists of three main components: BOTSMITH, TRANSLATOR, and SEARCHER. Given an MEV strategy, BOTSMITH generates an MEV bot in EVM bytecode (§ III-B). TRANSLATOR compiles the EVM bytecode to PTX code (referred to as the PTX bot), enabling parallel execution on the GPU. The two Solidity cheatcodes such as `vm.swap` and `vm.apply` are compiled into cross-contract calls in PTX to simulate the MEV transactions on GPUs. This transformation turns MEV search into an SIMD-parallel task, significantly improving throughput (§ III-C). TRANSLATOR runs only a single time, prior to the MEV search phase. Finally, SEARCHER employs a GPU-based genetic algorithm [14] to generate and evaluate bundles in parallel, selecting the most profitable candidates (§ III-D). Each generation produces a batch of offspring transactions, evaluated in parallel on the GPU. The most profitable bundle is replayed on a private Ethereum fork to validate correctness and eliminate false positives. In § V, we demonstrate `MeVisor`'s effectiveness by evaluating it on two classical MEV strategies: arbitrage and sandwich.

B. BOTSMITH

`MeVisor` is a wrapper of the Solidity compiler (*solc*). Given an MEV strategy written in Solidity, BOTSMITH compiles it into EVM bytecode and links the cheatcode functions. It eventually produces an MEV bot capable of executing bundles atomically. To assess profit through smart contract execution, the MEV bot must not only execute DEX transactions but also simulate the victim transactions.

However, transactions are ordered calls to EVM, but we want to discard EVM for higher throughput (**C1**). To simulate transactions without invoking the EVM, we introduce two key cheatcodes for writing MEV bots: `vm.swap` and `vm.apply` (see § IV-A). These primitive functions bypass EVM constraints and directly control smart contract states. `vm.swap` sends a smart contract call to a DEX to swap tokens. `vm.apply` simulates the victim transaction by executing its internal calls. The MEV bot leverages these cheatcodes to evaluate MEV profit. For an arbitrage attack, the bot uses a sequence of `vm.swap` calls to perform cyclic arbitrage swaps. In a sandwich attack, the bot first uses `vm.swap` to buy tokens, applies `vm.apply` to simulate the victim transaction, and then uses a second `vm.swap` to realize profit.

C. TRANSLATOR

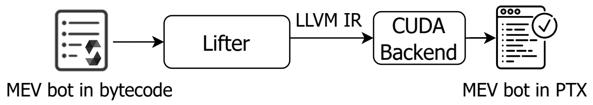


Fig. 5: Overview of TRANSLATOR.

As shown in Figure 5, once BOTSMITH generates the MEV bot, TRANSLATOR converts its EVM bytecode into PTX code using a two-stage compilation pipeline. First, TRANSLATOR lifts EVM bytecode to LLVM IR [40], enabling cross-contract calls by rewriting this architecture-independent representation (§ IV-B). Then, the LLVM backend compiles the IR into PTX code, which integrates with the CUDA application (i.e., SEARCHER), via library linking (§ IV-C).

TRANSLATOR operates on EVM bytecode, the universal compilation target of all EVM smart contract languages. Using EVM bytecode ensures language agnosticism, allowing DEXs implemented in different languages to run on the GPU. By recovering high-level semantics from low-level EVM bytecode, our approach ensures PTX compatibility with on-chain DEXs. Specifically, TRANSLATOR accelerates smart contract execution by converting EVM bytecode into SIMD instructions [48], allowing the same instruction to run across multiple data lanes in parallel. This process separates EVM instructions from input data, devirtualizes stack operations (e.g., push and pop) into register-based code in LLVM IR, and constructs a vectorized smart contract context for GPU execution. Each GPU thread maintains an isolated context, including stack, memory, storage, calldata and retdata, using thread-local addressing indexed by thread ID. To support cross-contract calls between MEV bots and DEXs, TRANSLATOR adopts function-call semantics with isolated call frames. However, TRANSLATOR faces a critical memory allocation challenge (**C2**): dynamically allocating per-thread smart contract contexts incurs hundreds of cycles per thread. This overhead increases significantly with deep cross-contract calls, as the EVM allocates a new context for each callee. To address this, TRANSLATOR includes a memory manager that uses pre-allocated shared memory for all smart contract contexts, eliminating dynamic allocation while maintaining isolation via call frames (§ IV-B3).

D. SEARCHER

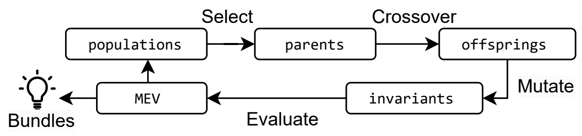


Fig. 6: The overview of SEARCHER.

Given the MEV bot's PTX code, SEARCHER executes it on the GPU to identify the most profitable bundle. As shown in Figure 6, SEARCHER implements a parallel genetic algorithm, where each individual in the population represents a candidate bundle in the search space (§ IV-D). In each generation, SEARCHER evolves the population through four steps: (1) selecting top-performing individuals as parents, (2) applying crossover to generate offspring bundles, (3) mutating transactions within the offspring to generate invariants, and (4) evaluating the profit in parallel. SEARCHER evaluates multiple populations concurrently by executing the MEV bot across GPU threads. This parallel genetic algorithm significantly improves throughput.

Input diversity plays a critical role in SEARCHER's performance. Naive mutations often produce transactions that violate MEV-specific constraints, hindering SEARCHER's throughput and search efficiency (**C3**). For example, cyclic arbitrage, which requires a sequence of DEX swaps that returns to the first token, is often disrupted by naive mutations. To address this, SEARCHER incorporates a DEX swap graph to guide mutation, ensuring that generated transactions adhere to valid MEV strategies (§ IV-D3).

IV. MeVisor Design

This section outlines the technical details of MeVisor.

A. Cheatcode for MEV Strategies

We design two cheatcodes in § IV-A1 and illustrate their uses in MEV strategies in § IV-A2.

1) Cheatcode: We design two primitive functions, `vm.swap` and `vm.apply`, as cheatcodes compiled into PTX alongside the MEV bot. These cheatcodes enable GPU-based execution of arbitrage and sandwich MEV.

vm.swap accepts a DEX address, swaps a pair of tokens on the specified DEX, and returns the amount of the output token. This cheatcode enables direct cross-contract calls, bypassing router contracts. In doing so, it reduces execution overhead and facilitates efficient cross-protocol swaps.

vm.apply takes a transaction hash and simulates execution of the corresponding transaction. When BOTSMITH encounters a `vm.apply(tx_hash)` call, it extracts the internal cross-contract calls from the transaction identified by `tx_hash` and applies them to the MEV bot for execution. This cheatcode allows the MEV bot to front-run or back-run transactions as specified by MEV strategies. For instance, by placing token swaps immediately before or after `vm.apply(tx_hash)`, the bot can launch a sandwich attack.

2) MEV Strategies: Building on the cheatcodes introduced in § IV-A1, this section presents two representative MEV strategies that demonstrate MeVisor's capabilities:

Arbitrage exploits price discrepancies across multiple DEXs. In this paper, we only consider cyclic arbitrage. A cyclic arbitrage consists of a sequence of swap trades. In each trade, $R^{w}_{i,j}$ denotes the exchange rate of token i over j on DEX w. Given the amount of the input token Q, the total token output from an arbitrage MEV with n trades is $Q * R^{w_0}_{0,1} * R^{w_1}_{1,2} * \cdots * R^{w_{n-1}}_{n-1,n}$. Then, the revenue is calculated as the total output value minus the input value:

$$Q(\prod_{i=0}^{n-1} R^{w_i}_{i,i+1} - 1)$$

The profit is then calculated as Revenue $-\ C$, where C represents the total gas fee. Cyclic arbitrage is implemented as a sequence of calls to `vm.swap`.

Sandwich targets a large, price-moving transaction (often referred to as a whale transaction) and exploits the predictable slippage it induces. A typical sandwich attack consists of three sequential transactions: $[R^{w_1}_{ij}, \mathbb{R}^{w_2}_{ij}, R^{w_3}_{ji}]$, where: 1) *front-run:* Swap token A_i for token A_j on DEX w_1, intentionally inflating the price of A_j due to whale transaction. 2) *whale-sim:* Simulate the whale transaction using `vm.apply`. BOTSMITH constructs a transaction equivalent to $q \cdot \mathbb{R}^{w_2}_{ij}$, where the victim expects at least δ units of A_j are swapped from q units of A_i on DEX w_2. If the simulation fails (i.e., $q \cdot \mathbb{R}^{w_2}_{ij} < \delta$), the MEV attack is unsuccessful. 3) *back-run:* Swap token A_j back to token A_i on DEX w_3, after the successful simulation of the whale transaction. The revenue from a cyclic sandwich attack is given by:

$$\mathbb{I}[q \cdot \mathbb{R}^{w2}_{ij} \geq \delta] \cdot Q \cdot (R^{w1}_{ij} \cdot R^{w2}_{ji} - 1)$$

Here, $\mathbb{I}[\cdot]$ is an indicator function that evaluates to 1 if the whale transaction succeeds (i.e., $q \cdot \mathbb{R}^{w_2}_{ij} \geq \delta$), and δ is the minimum expected output of the whale transaction. Similarly, the final profit is Revenue $-\ C$, where C is the total gas fee. In sandwich attacks, only the DEX-related executions of the whale transaction are simulated on the GPU, as they constitute the primary source of slippage.

B. Translating Bytecode to IR

Once the MEV bot is built, TRANSLATOR translates its EVM bytecode into LLVM IR [40] through three steps: First, it generates LLVM function headers that correspond to the EVM bytecode of the smart contracts (§ IV-B1). Second, it constructs function bodies by lifting EVM instructions into LLVM IR using devirtualization techniques (§ IV-B2). Finally, it identifies cross-contract calls within the EVM bytecode and translates them into virtual function calls (§ IV-B3). Note that support for cross-contract calls is essential, as token swaps inherently require interactions across multiple DEXs.

1) Building Function Headers: Each smart contract's bytecode is translated into a distinct LLVM function, with the function scope using as the boundaries of the smart contract context. The function name is uniquely derived from a hash of the EVM bytecode. Function arguments include EVM-specific inputs, such as the sender address (`msg.sender`) and transferred value (`msg.value`), along with pointers to EVM components: calldata (d), stack (μ), memory (v), retdata (r), and storage (θ). These arguments are locally scoped to preserve smart contract context isolation between caller and callee during cross-contract calls. In contrast, block-level information and the original sender of the transaction are modeled as global variables, since the EVM exposes them uniformly across all call frames.

To support efficient SIMD execution, each EVM component is extended into a vector structure, where each element is mapped to a distinct GPU thread. Input data and output buffers are partitioned evenly across threads based on their thread indices. If thread i receives input from $\vec{d_i}$ and computes an MEV profit $\vec{r_i}$, then the parallel execution is modeled as: $\vec{r_i} = I \times \vec{d_i} \times \vec{\nu_i} \times \vec{\mu_i} \times \vec{\theta_i}$, where each instruction I operates simultaneously across multiple data lanes and performs an element-wise SIMD execution. Each thread operates exclusively on its own thread-local data, reducing contention and synchronization overhead. Furthermore, we adopt a Structure of Arrays (SoA) layout for EVM components, storing thread-specific data in contiguous memory locations. This design facilitates coalesced memory access [13], enabling multiple threads to read or write data simultaneously and thereby improving throughput.

2) Lifting for LLVM IR: The EVM employs a stack-based execution model, in which instructions implicitly manipulate a global operand stack. To lift EVM bytecode into register-based LLVM IR, we adopt the devirtualization technique proposed in [9], which simulates the operand stack and transforms all stack operations into explicit memory operations. Specifically, each EVM stack element is mapped to a temporary register, thereby exposing implicit data dependencies.

A local variable p is used to track the EVM stack depth. For example, in the `PUSH` opcode, which pushes a value onto the stack, the value is assigned to $\vec{\mu}_p$, and p is incremented. In the `POP` opcode, which removes a value from the stack, the value is read from $\vec{\mu}_p$, and p is decremented accordingly. By converting implicit stack manipulations into explicit register assignments, we enable data-flow analysis and eliminate redundant operands. For instance, the `MUL` opcode pops two operands from the stack, multiplies them, and pushes the result back. In our SIMD translation, this operation becomes:

$$\vec{\mu}_{p-1} \leftarrow \vec{\mu}_{p-1} \times \vec{\mu}_p; p \leftarrow p - 1$$

This formulation allows direct use of labeled registers, removing the need for runtime operand resolution required by the EVM. In addition to arithmetic instructions, control-flow operations must also be translated to support parallel execution. Since PTX branch instructions are inherently thread-safe, intra-contract control flow (i.e., conditional branches) is lifted directly into PTX branching instructions such as `bne`. Cross-contract calls, commonly invoked when the MEV bot interacts with DEXs, are handled separately and translated into LLVM function calls, as discussed in § IV-B3.

3) Cross-contracts Calling: In the EVM, the targets of cross-contract calls are determined at runtime based on values stored on the operand stack. To recover these implicit control transfers in LLVM IR, we translate cross-contract calls into virtual function calls. We construct a virtual table that contains the addresses of all potential smart contract functions, indexed by their unique code hashes. Each entry in the table points to an LLVM function translated from the corresponding EVM bytecode. At each call site, the EVM call instruction is replaced with a virtual function call that dynamically resolves the target contract via the virtual table. To reduce the overhead of dynamic dispatch, we apply static analysis to resolve call targets whenever possible. In such cases, the virtual call is replaced with a direct LLVM function call. This optimization is particularly effective for frequent interactions between the MEV bot and DEX contracts.

Each smart contract is compiled into an individual LLVM function; thus, the LLVM call frame naturally serves as the execution context for cross-contract calls. In the EVM, every such call allocates a fresh stack and memory, which are deallocated upon return. While dynamic memory allocation can simulate this behavior, it is prohibitively expensive on GPUs. To address this, we statically allocate memory and overload EVM components across all call frames while maintaining isolation between them. We construct isolated call frames for cross-contract execution in three stages, as described below.

call_begin. Each GPU thread maintains a thread-local register q, initialized to zero, to track the current call depth. Before executing a cross-contract call, the call depth q increases by one. Each call frame at depth q accesses its isolated context using a fixed offset within the statically allocated memory region. This offset is represented as $\vec{c}_{i \cdot m_c \cdot q}, \quad c \in \{\mu, \nu\}$, where m_c is the size of the corresponding component, and i is the thread index.

call_run. To perform the cross-contract call, control is transferred via a lookup in the virtual table. The callee address is resolved, and the corresponding function pointer in the table is invoked. The callee's memory and stack, corresponding to a new call frame, are passed as function arguments. Additional function arguments, such as `msg.sender` and `msg.value`, are populated based on the EVM call type (`CALL`, `STATICCALL`, `CALLCODE`, or `DELEGATECALL`). Readers may read the Ethereum Yellow Paper [62] for the specification of call semantics.

call_end. After the callee completes execution, the call depth q is decremented to return control to the caller. If the EVM call returns data, the result is copied from the callee's memory space back into the caller's memory.

C. Driving PTX

To execute the MEV bot on the GPU, we generate valid PTX code from LLVM IR (§ IV-C1) and implement a driver that executes smart contracts within the CUDA environment (§ IV-C2).

1) Generating PTX Code: To emit PTX code using the LLVM backend, we embed a set of PTX-specific primitives directly into the LLVM IR.

Kernel Function. In PTX, functions are categorized as either *kernel functions* or *device functions*. Kernel functions act as entry points for host code to launch GPU execution. By default, functions defined in LLVM IR are treated as device functions and therefore cannot be invoked directly from the host. To enable GPU execution, we make the MEV bot's entry function to be a kernel function by attaching a metadata node, `nvvm.annotations`, which assigns the "kernel" attribute to the function [50].

CUDA Registers. CUDA-specific registers, such as thread indices, are accessed via LLVM intrinsics during bytecode translation [49]. Intrinsics are built-in LLVM functions that expose low-level hardware features within the IR. During the lifting process, TRANSLATOR inserts these intrinsics to retrieve thread-specific data. For example, the thread index used to access each thread's SIMD input. The LLVM backend then compiles these intrinsics into corresponding PTX instructions that interact directly with GPU hardware.

CUDA Memory. The CUDA memory model defines multiple memory spaces on the GPU, each with distinct physical locations, lifetimes, and performance characteristics. In LLVM IR, these spaces are explicitly annotated using address space metadata, which guides the PTX backend in mapping high-level memory abstractions to appropriate GPU hardware re-

Algorithm 1 Genetic algorithm for search MEV opportunities

```
Input:
  $Bot$: the MEV bot in PTX
  max_rounds: Maximum rounds
Output: Best transaction to the MEV bot
1: $\vec{P}, \vec{M} \leftarrow DryRun(Bot)$
2: $i \leftarrow 0$
3: while $i++ <$ max_rounds do                    ▷ searching loop
4:     $\vec{X}, \vec{Y} \leftarrow TournamentSelect(\vec{M})$
5:     $\vec{P'} \leftarrow Crossover(\vec{P}, [\vec{X}, \vec{Y}])$
6:     $\vec{P'} \leftarrow Mutate(G, \vec{P'})$
7:     $\vec{M} \leftarrow Evaluate(Bot, \vec{P'})$
8: end while
9: Return: $\vec{P'}_{argmax(\vec{M})}$
```

sources. We leverage this model to manage data efficiently for smart contract execution. Calldata, which varies across transactions, is allocated in global memory to support mutation. In contrast, static inputs, such as the caller's address and blockchain environment variables, are placed in constant memory. Constant memory provides immutability and lower latency after initialization, making it well-suited for read-only data shared across GPU threads.

2) Binding PTX and CUDA: To enable CUDA applications (i.e., SEARCHER) to invoke the PTX bot compiled from Solidity, we embed the PTX code into the CUDA executable. We use CUDA's low-level driver APIs to manage memory and resolve external symbols such as kernel functions and global variables within the PTX module. These symbols are then exposed through well-defined interfaces, facilitating seamless integration and invocation from CUDA host code.

PTX embedding. To integrate PTX with CUDA, we first create a GPU context using the CUDA Driver API function `cuCtxCreate()`. The CUDA application (i.e., SEARCHER) then loads the PTX module onto the GPU using `cuModuleLoad()`. This newly created context replaces the default one, allowing PTX and CUDA code to operate within a unified address space for both code and memory.

Code Space Integration. PTX kernel functions are declared with external visibility, which allows CUDA applications to retrieve their function pointers. SEARCHER uses these pointers to launch PTX bot executions efficiently across GPU threads.

Memory Space Integration. We resolve GPU memory addresses referenced by the PTX code and manage data movement using CUDA's host-driven APIs, including both host-to-device and device-to-device memory transfers.

D. Searching with Genetic Algorithm

Algorithm 1 presents our genetic algorithm, designed to efficiently search for profitable MEV bundles. Given a PTX bot Bot and a maximum number of iterations max_rounds, the goal is to identify the transaction bundle that yields the highest profit. Each token swap is modeled as a gene, and an individual consists of $\mathbb{N}$ genes, where $\mathbb{N}$ denotes the maximum number of swap actions permitted in an MEV transaction. The population $\vec{P}$ is a vector of candidate individuals serving as input to the genetic algorithm. Each individual represents a transaction bundle to be optimized. Fitness values are evaluated using the MEV bot and stored in the vector $\vec{M}$, where each entry corresponds to the profit of an individual. The algorithm begins with a randomly generated population (referred to as a *DryRun*), which is executed by the MEV bot to establish initial fitness scores. Subsequently, the population evolves through four genetic operations: selection (§ IV-D1), crossover (§ IV-D2), mutation (§ IV-D3), and fitness evaluation (§ IV-D4). After all generations complete, the most profitable bundle, denoted as $\vec{P'}_{argmax(\vec{M})}$, is replayed on a CPU-side EVM to verify correctness and eliminate false positives.

1) Parallel Selection: We implement a tournament selection [47] on the GPU to efficiently identify high-quality individuals from the current population. These selected individuals serve as parents for generating the next generation. For each selection, we randomly sample κ individuals from the $\vec{P}$ and select the individual with the highest fitness, storing its index in $\vec{I}$. Formally, for each tournament, we compute: $I \leftarrow \arg\max(\{\vec{M}_i \mid i \in \text{rand}, 1 \leq i \leq |\vec{P}|\})$, where $\vec{M}$ denotes the fitness scores of all individuals, and i is a randomly selected subset of indices of size κ. To parallelize the selection process, we create two arrays, X and Y, to store the indices of selected parents from $\vec{P}$. We launch $|\vec{P}|$ GPU threads, where each thread independently performs two tournament selections to populate one entry in X and Y. This fully parallel approach maximizes throughput by exploiting the massive concurrency available on the GPU.

2) Parallel Crossover: Next, crossover is performed between pairs of selected transaction candidates (i.e., parents) to generate offspring for the next generation. Each offspring $\vec{P'}_i$ is produced by combining genes from two parents, $\vec{P}_{X_i}$ and $\vec{P}_{Y_i}$, whose indices are drawn from the arrays X and Y, respectively. Formally, the operation is defined as: $\vec{P'}_i \leftarrow \vec{P}_{X_i} \times \vec{P}_{Y_i}$, where $\times$ denotes the crossover operator applied to two MEV bundles. The crossover operation typically exchanges subsequences of swap actions between parents, allowing offspring to inherit profitable MEV paths from both lineages. This design enables efficient, thread-level parallelism, as each GPU thread independently generates a new offspring, maximizing throughput in the candidate generation phase.

3) Graph-based Swap Mutation: Next, we apply mutation to the offspring population $\vec{P'}$ to introduce variation and preserve genetic diversity. Each individual in the population encodes a sequence of swap actions. Mutation is performed to capital and sequence. **Capital mutation** modifies the input token amount (a `uint256` value) using fuzzing-inspired strategies from AFL [34], such as Bitflip, Byteflip, and Havoc. These mutations promote numeric diversity in transaction inputs. **Sequence mutation** changes the MEV paths involved in a bundle. Since MEV bots often impose structural constraints, unconstrained mutations may produce invalid or unprofitable bundles, leading to wasted GPU cycles. For example, arbitrage transactions require cyclic swap structures. To guide the sequence mutation, we construct a swap graph G, where each edge $G_{i,j}$ denotes the set of DEXs that support swaps between the token pair $[A_i, A_j]$. Leveraging this graph, SEARCHER applies the following mutation strategies:

- **Add.** Insert an intermediate token to extend an MEV path. Given a swap sequence $[A_i, A_j]$, insert a random token A_k to have $[A_i, A_k, A_j]$. The new path is considered valid if both $G_{i,k} \neq \emptyset$ and $G_{k,j} \neq \emptyset$.
- **Remove.** Remove an intermediate token from the MEV path. Given a sequence $[A_i, A_k, A_j]$, remove A_k to obtain $[A_i, A_j]$. If $G_{i,j} \neq \emptyset$, the simplified path remains valid.
- **Replace.** Replace an intermediate token with a new one. For example, transform $[A_i, A_k, A_j]$ into $[A_i, A_m, A_j]$ by replacing A_k with a randomly selected token A_m. The modified path is valid if $G_{i,m} \neq \emptyset$ and $G_{m,j} \neq \emptyset$.

4) Fitness Evaluation: To identify high-profit individuals within the current population $\vec{P}$, we evaluate their fitness by executing the MEV bot in parallel across GPU threads. During this evaluation, the GPU reconstructs smart contract states, including the amount of token reserves and liquidity, to simulate realistic DEX conditions. This enables practical assessment of MEV opportunities in a setting that closely mirrors mainnet execution. Fitness is quantified as the transaction revenue (in ETH), computed according to the strategies defined in § IV-A2. The resulting fitness values are stored in the vector $\vec{M}$ and subsequently used in the next evolutionary round for selection and crossover.

5) Forking Mainnet: To validate MEV outcomes, we construct a simulation environment by forking the Ethereum mainnet using Foundry [32]. This forked environment is used exclusively for final MEV validation after candidate bundles are discovered on the GPU.

Although standard RPC endpoints allow forking from a specific block number, they do not support rollback to the pre-state of an arbitrary transaction. To reconstruct the precise blockchain state immediately before a target transaction, we retrieve all preceding transactions within the block via `web3.eth.get_block()` and replay them sequentially via `web3.eth.send_raw_transaction()`.

E. Implementation

We implement a prototype of `MeVisor` using 2,541 lines of C/C++ and 2,007 lines of CUDA. In addition to compiling `MeVisor` as a standalone executable, we also compile it as a dynamic library accessible via Python bindings, using `pybind11` [53]. This Python integration facilitates efficient GPU-accelerated MEV search and enables seamless interaction with Ethereum clients for transaction validation via Web3.py [22]. Appendix B shows the detailed novelty of `MeVisor` against [9]. Below, we summarize the core components:

BOTSMITH. BOTSMITH is a wrapper around `solc` [25], responsible for injecting the cheatcode library into the source code of the MEV bot and compiling it with DEXs into EVM bytecode.

TRANSLATOR. TRANSLATOR is a compiler built on the LLVM framework that supports smart contract semantics. In particular, it supports to translate cross-contract calls on the GPU. Each GPU thread is provisioned with resources sufficient for typical DEX executions: 512 stack items, 724 bytes of memory, and 2 storage slots.

SEARCHER. SEARCHER implements a parallel genetic algorithm entirely in CUDA. To improve throughput, we launch 8,192 GPU threads concurrently. The initial population is generated using a depth-first search algorithm, and tournament selection is configured with a tournament size of $\kappa = 256$.

TABLE II: Profile of the ground-truth benchmark. #Num, #DEX, and #Token denote the number of unique attacks, decentralized exchanges, and tokens, respectively. $Avg. denotes the average value.

	#Num	#DEX	#Token	$Avg.Revenue	$Avg.Cost	$Avg.Profit
Arbitrage	2,380	1,201	504	1217.84	619.88	597.97
Sandwich	1,561	1,010	942	677.59	169.33	508.26
Total	3,941	2,009	1,271	1,003.85	441.42	562.43

V. EVALUATION

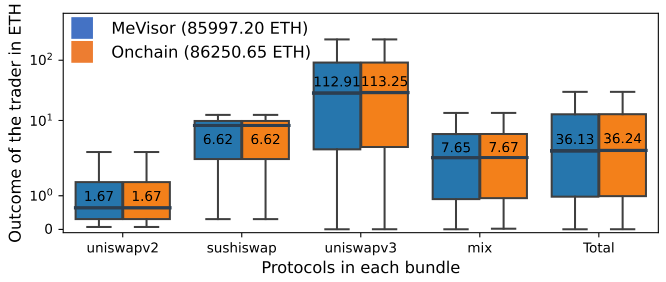
Ground-Truth Benchmark. We construct a comprehensive ground-truth benchmark of real-world MEV activity. An Ethereum archive node is deployed using Erigon [20] to index all transactions from 2023. Ether is priced using historical market data [23]. We extract MEV attacks with on-chain profits exceeding 100 USD, yielding a total of 3,941 attacks, including 2,380 arbitrage attacks and 1,561 sandwich attacks. For each attack, we record: (1) the block number in which the attack occurred, (2) the transactions involved, and (3) on-chain revenue and profit obtained from EigenPhi [18]. The dataset spans 1,271 unique tokens and 2,009 DEXs, including major protocols such as SushiSwap, Uniswap V2, and Uniswap V3, together accounting for over 70% of MEV-related DEX interactions on Ethereum [45]. In total, these attacks generated \$3.76 million in gross revenue and \$2.21 million in net profit.

Baseline Config. Lanturn [5] is the only non-deterministic tool for MEV search. We adopt its default configuration from the artifacts and apply the following modifications to ensure a fair and realistic evaluation: First, we revise Lanturn's profit calculations. Lanturn assumes that the attacker can mine blocks and therefore includes a 2 ETH block reward as part of the MEV profit. We exclude this reward from our evaluation, as real-world MEV searchers do not rely on mining privileges. Second, Lanturn assumes attackers have sufficient time to swap tokens on off-chain markets (i.e., Binance). Accordingly, we also disable it because we only focus on on-chain profit. For performance benchmarking, we execute Lanturn with 44 parallel CPU threads.

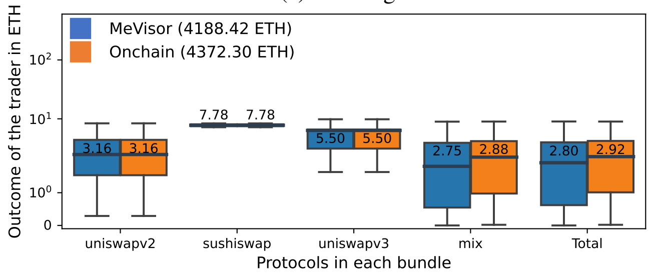
Environment Setup. All experiments were conducted on a server running Ubuntu 20.04 LTS, equipped with an Intel Xeon processor, 64 GB of RAM, a 1 TB SSD, and an NVIDIA RTX 3090 GPU with 24 GB of VRAM.

Research Questions. Our evaluation is guided by the following research questions (RQs):

- **RQ1**: Are MEV bots generated by TRANSLATOR sound in reproducing known MEV attacks on the GPU?
- **RQ2**: How does `MeVisor` compare to baseline approaches in terms of effectiveness, efficiency, and throughput?
- **RQ3**: How much profit can `MeVisor` uncover when deployed in a real-world setting?



(a) Arbitrage



(b) Sandwich

Fig. 7: Replay results of MEV bundles across different DEX protocol combinations.

A. Soundness

Motivation. Execution inconsistencies introduced by TRANSLATOR may reduce the soundness of MEV detection. Underestimating profits on the GPU can lead to false negatives, causing MEV opportunities to be missed. Conversely, overestimating profits may result in false profit. We aim to quantify the impact of these inconsistencies and assess whether the smart contracts generated by TRANSLATOR can maintain soundness.

Approach. We conduct differential testing by: (1) forking a blockchain environment, (2) disabling non-deterministic search, (3) replaying historical MEV bundles on both the GPU and CPU, and (4) comparing the resulting output tokens valued in ETH. We define GPU execution accuracy as U'/U, where U' and U represent the output tokens computed by the GPU and CPU, respectively.

Overall Results. TRANSLATOR successfully compiles all DEX contracts and executes them on the GPU. Over all arbitrage and sandwich attacks, GPU execution yields 90,185.62 ETH in output, closely aligning with the CPU result of 90,622.96 ETH, achieving an execution accuracy of 99.52%.

TABLE III: Arbitrage replay results.

	#Num	#Same	Total Outcome/ETH		
			Accuracy	GPU	CPU
Uniswap V2	86	86	100%	143.46	143.46
Sushiswap	6	6	100%	39.73	39.73
Uniswap V3	649	154	99.70%	73281.16	73500.54
Mix	1639	904	99.73%	12532.84	12566.92
All	2380	1149	99.71%	85997.20	86250.65

These results demonstrate that TRANSLATOR is sound. All observed inconsistencies stem from minor underestimations on the GPU, which may lead to false negatives but never to false positives.

Results of Arbitrage MEV. Table III shows the replay results of 2,380 arbitrage bundles, each involving two to five transactions. Overall, `MeVisor` recovers 85,997.20 ETH of output token (equivalent to 142.90 million USD), achieving 99.71% of the value observed on-chain. Across all arbitrage cases, 48.28% of bundles have a 100% execution accuracy. The remaining cases show only slight conservative underestimations on the GPU.

To better understand the inconsistency of GPU execution, we group the replay results by DEX protocol, as DEXs within the same protocol typically follow a common codebase. We observe that GPU smart contracts perfectly reproduce all arbitrage swaps involving only Uniswap V2 or Sushiswap DEXs. Specifically, all 86 Uniswap V2 and 6 Sushiswap arbitrage bundles yield identical outcomes on the GPU and CPU, recovering 143.46 ETH and 39.73 ETH, respectively. For Uniswap V3, the accuracy is slightly lower at 99.70%, with 73,281.16 ETH recovered on the GPU versus 73,500.54 ETH on the CPU. Out of 649 bundles involving only Uniswap V3 DEXs, 23.7% of them produce the same outcomes, while the remaining 495 show minor discrepancies. These discrepancies arise from an intentional GPU-side simplification: disabling "tick spacing," a slippage-control mechanism in Uniswap V3, to reduce GPU memory pressure. In practice, "Tick Spacing" is not always necessary, as most MEV opportunities involve a small capital input that does not trigger slippage. This explains why 1,058 of 1,639 bundles involving Uniswap V3 still maintain high accuracy, resulting in an overall accuracy of 99.73%. We further show the distribution of token outputs from GPU and CPU executions in Figure 7a. Despite the simplification, only minor deviations are introduced in Uniswap V3 swaps, with median outcomes of 112.91 ETH (GPU) vs. 113.25 ETH (CPU). In the worst case, GPU execution underestimates the DEX output by up to 3.20%. Since SEARCHER always aims to maximize output, any deviation can only lead to underestimation, thereby ensuring no false positives are introduced.

Results of Sandwich MEV. Similarly, Table IV presents the replay results of 1,561 sandwich bundles, each consisting of two swaps surrounding a whale transaction. In total, `MeVisor` recovers 4,188.42 ETH, equivalent to 7.65 million USD, achieving 95.79% of the profit recorded on-chain (4,372.30 ETH). A total of 128 sandwich bundles are perfectly replayed

TABLE IV: Sandwich replay results.

	#Num	#Same	Total Outcome/ETH		
			Accuracy	GPU	CPU
Uniswap V2	115	115	100%	363.95	363.95
Sushiswap	2	2	100%	15.56	15.56
Uniswap V3	5	5	100%	27.50	27.50
Mix	1433	6	95.36%	3781.41	3965.29
All	1561	128	95.79%	4188.42	4372.30

on the GPU, achieving 100% accuracy. Sandwich bundles involving only one kind of DEX protocol demonstrate full consistency between GPU and CPU execution. Among these, all 115 Uniswap V2, 2 Sushiswap, and 5 Uniswap V3 bundles yield identical outcomes, recovering 363.95 ETH, 15.56 ETH, and 27.50 ETH, respectively. The majority of sandwich attacks (1,433 bundles, or 91.8%) span multiple DEXs. In these more complex cases, `MeVisor` achieves 95.36% accuracy, recovering 3,781.41 ETH on the GPU compared to 3,965.29 ETH on the CPU, with a deviation of only 4.63%. The median per-bundle profit is slightly underestimated (2.75 ETH vs. 2.88 ETH), as shown in the boxplots of Figure 7b.

Unlike arbitrage, sandwich execution requires faithfully modeling the victim whale's transaction on the GPU, as this trade introduces price slippage that attackers exploit. To reduce GPU complexity, `MeVisor` approximates the whale transaction by replaying only its swap transactions. This approximation affects the simulation of post-whale price dynamics, particularly in multi-DEX scenarios where market effects interact. As a result, the attacker's post-whale swap may execute at a slightly different price, leading to a conservative underestimation of the final MEV outcome. Despite these deviations, the approximation strategy remains sound for MEV search, as all bundles are validated on the CPU.

Answer to RQ1: TRANSLATOR reliably generates MEV bots capable of soundly replaying real-world MEV attacks on the GPU.

B. Comparison with Baselines

Motivation. To assess the performance of `MeVisor`, we compare it against the state-of-the-art baseline, Lanturn. We observe that Lanturn requires a template input from known transactions to start its searching. For a fair comparison, we design an experiment in which both tools attempt to front-run historical MEV transactions. We define the revenue ratio as the revenue obtained by the tool divided by the on-chain revenue. A ratio greater than one indicates successful front-running, meaning the tool identifies a transaction more profitable than the one originally executed on-chain.

Approach. We generate Lanturn's input using the Python scripts provided in its public artifact. For arbitrage, the transaction template includes: (1) a list of approved ERC-20 tokens, (2) the swap sequence extracted from historical MEV attacks, (3) parameterized fields for token amounts, and (4) the input range over which Lanturn performs its search. For sandwich attacks, the template additionally includes the whale transaction to mock the victim. Lanturn then optimizes each template to maximize profit. For `MeVisor`, historical MEV attacks serve as initial inputs, which are further optimized using a GPU-based genetic algorithm. Note that historical inputs are not constrained by `MeVisor`, as the initial bundles also include randomly constructed ones. In addition to revenue, we measure profit, gas cost, and transaction throughput to evaluate overall performance.

TABLE V: Comparison against Lanturn on arbitrage MEV.

	ETH			USD		
	Profit	Revenue	Cost	Profit	Revenue	Cost
`MeVisor`	1842.78	2167.67	324.89	3232556.99	3784195.81	551638.82
On-chain	802.42	1669.03	866.61	1423160.32	2898468.92	1475308.60
Lanturn	817.64	1685.05	867.40	1451491.27	2928380.11	1476888.85

TABLE VI: Comparison against Lanturn on sandwich MEV.

	ETH			USD		
	Profit	Revenue	Cost	Profit	Revenue	Cost
`MeVisor`	490.36	582.35	92.00	896449.29	1064815.60	168366.31
On-chain	435.10	578.59	143.49	793394.00	1057722.97	264328.97
Lanturn	450.44	578.59	128.15	819939.37	1057722.97	237783.60

MEV Revenue. Overall, `MeVisor` outperforms Lanturn, achieving up to 1.21x higher MEV revenue. `MeVisor` yields a total of 2,167.67 ETH in arbitrage revenue, corresponding to 129.88% of the original on-chain value. In contrast, Lanturn achieves only 100.96% of the on-chain revenue, indicating that `MeVisor` extracts 1.30x more arbitrage revenue than Lanturn. Similarly, we present the results for sandwich attacks. `MeVisor` generates 582.35 ETH in revenue (100.65% of the on-chain value), while Lanturn remains at a revenue ratio of 1.00 and fails to uncover any additional profit during front-running. These findings demonstrate that `MeVisor` consistently outperforms Lanturn in MEV revenue extraction across both arbitrage and sandwich strategies.

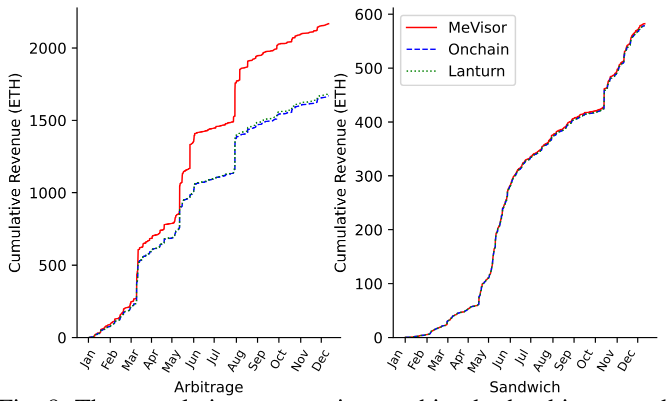


Fig. 8: The cumulative revenue in searching both arbitrage and sandwich opportunities.

Revenue Trending. We further measure the cumulative MEV revenue over time. `MeVisor` consistently achieves higher cumulative MEV revenue than both the on-chain baseline and Lanturn, demonstrating superior front-running capabilities

TABLE VII: Throughput comparison against Lanturn.

	Arbitrage			Sandwich		
	time	bundles/sec	txs/sec	time	bundles/sec	txs/sec
MeVisor	11.49	1,425.93k	3,378.49k	9.49	1,726.52k	5,179.57k
Lanturn	115.90	5.83	30.35	14.97	7.79	23.37

over time. Figure 8 illustrates the cumulative MEV revenue over the year for both arbitrage and sandwich attacks. For arbitrage attacks, the on-chain baseline, Lanturn, and MeVisor yield revenues of 1,669.03 ETH (2,898,468.92 USD), 1,685.05 ETH (2,928,380.11 USD), and 2,167.67 ETH (3,784,195.81 USD), respectively. For sandwich attacks, both the on-chain baseline and Lanturn yield 578.59 ETH (1,057,722.97 USD), while MeVisor slightly improves upon this with 582.35 ETH (1,064,815.60 USD).

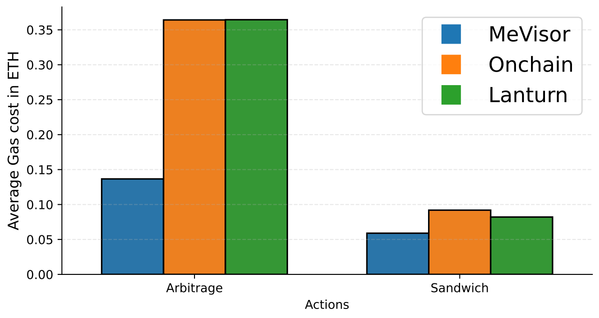


Fig. 9: The average gas cost in searching both arbitrage and sandwich opportunities.

Gas Cost. Practical MEV attacks must account for gas costs to ensure profitability. To ensure a fair comparison, all transactions are replayed on the same forked blockchain, allowing gas fees to be measured under identical conditions. Figure 9 depicts the gas cost of both arbitrage and sandwich results. On average, MeVisor incurs a gas fee of 0.11 ETH per MEV bundle, representing only 41.88% of Lanturn's average cost. For arbitrage attacks, the average gas fees for MeVisor, on-chain attackers, and Lanturn are 0.14 ETH (231.78 USD), 0.36 ETH (619.88 USD), and 0.37 ETH (620.54 USD), respectively. For sandwich attacks, MeVisor consumes only 0.06 ETH (107.86 USD) per bundle, compared to 0.09 ETH (169.33 USD) for historical attackers and 0.08 ETH (152.33 USD) for Lanturn. MeVisor achieves lower gas costs by issuing direct calls to DEX contracts, thereby avoiding the overhead associated with router-based execution, as used in Lanturn.

MEV Path. We observe that MeVisor significantly outperforms Lanturn as the length of the arbitrage path increases. Lanturn fails to exceed on-chain revenue for arbitrage paths involving more than two swaps. In contrast, MeVisor remains effective across longer paths; for example, it achieves up to 1.0945x higher revenue on five-hop arbitrage paths.

Time. MeVisor also outperforms Lanturn in search time, making it practical for real-time MEV execution. Table VII reports the average time required to identify MEV opportunities. MeVisor completes arbitrage and sandwich searches in 11.49 seconds and 9.49 seconds on average, respectively. Both are within Ethereum's 12-second block interval. In contrast, Lanturn requires 115.90 seconds for arbitrage and 14.97 seconds for sandwich searches.

TABLE VIII: MEV opportunities identified by MeVisor in 2025 Q1 blocks.

	ETH			USD		
	Profit	Revenue	Gas	Profit	Revenue	Gas
Arbitrage	335.53	336.40	0.87	884,116.65	886,409.48	2,292.83
Sandwich	90.90	103.60	12.70	239,525.84	272,975.41	33,449.57
Total	426.43	440.00	13.57	1,123,642.49	1,159,384.89	35,742.40

Throughput. As shown in Table VII, MeVisor achieves an average throughput of 1.43M bundles/sec on the arbitrage dataset and 1.73M bundles/sec on the sandwich dataset, which is over 100,000x faster than Lanturn. In comparison, Lanturn processes only 5.83 and 7.79 bundles/sec on the same datasets, respectively. We observe the higher throughput on sandwich attacks because arbitrage bundles typically contain more transactions that must be executed. We further evaluate transaction-level throughput: MeVisor achieves 3,378.49K txs/sec for arbitrage and 5,179.57K txs/sec for sandwich searches, significantly outperforming Lanturn. Lanturn's performance bottleneck primarily stems from its use of a slower, TypeScript-based EVM [35]. Although switching to a faster engine such as the Rust-based Revm [7] could improve performance, it remains substantially slower than MeVisor. Moreover, MeVisor surpasses even the fastest known GPU-based EVM [9], which peaks at approximately 300K txs/sec. This performance advantage arises from TRANSLATOR, which compiles smart contracts into GPU executables rather than interpreting them using the EVM. In addition, MeVisor disables rollback, a feature required for EVM correctness but unnecessary for MEV searchers who discard failed transactions, which significantly reduces data dependencies and computational overhead on GPUs.

Answer to RQ2: MeVisor outperforms the state-of-the-art baseline in terms of effectiveness, efficiency, and throughput.

C. *MEV found by MeVisor*

Motivation. To evaluate the effectiveness of MeVisor in uncovering MEV opportunities, we applied it to 648,000 Ethereum mainnet blocks collected over a 90-day period from January to March 2025 (Q1).

Approach. To identify cross-DEX profit opportunities, we collected data on 454 distinct tokens across 540 popular DEXs, as listed by CoinGecko [33]. MeVisor takes their on-chain prices as input. SEARCHER was configured with a 12-second timeout to reflect Ethereum's real-world block interval. We set the maximum transaction path length to 16, as longer paths incur higher gas fees that often outweigh the potential profit. To estimate the financial impact, we use the following setting.

TABLE IX: Distribution of arbitrage MEV by bundle length.

#Path	#Cnt	$Revenue	$Profit	#Path	#Cnt	$Revenue	$Profit
2	210	87,411.53	85,960.53	10	2	188.92	143.43
4	1	753,444.85	753,407.19	12	3	37,431.54	37,051.44
6	2	3799.68	3771.49	13	3	2,695.28	2,523.58
8	5	260.79	208.98	14	3	1,176.90	1,050.01

Price of Ether: USD 2,635/ETH, which was fixed at its market value on May 28, 2025. **Slippage**: The attacker always uses all input tokens to maximize token profit, thus disabling slippage thresholds (i.e., `minAmountOut` = 0). As for the sandwich victim, his/her slippage is preserved from historical transactions to mock the victim's trade. **Fee-on-transfer (tax) tokens**: The attacker receives the same output token as in the original MEV incident and pays gas fee in ETH. **Gas Model**: The base fee is taken from historical block data, while the priority fee is set to match the average bids in the same block [51]. **Cancellation costs**: Since our model assumes a single searcher without competition, failed transactions are not counted. To ensure correctness, all bundles were validated on a private forked chain. As validation occurred off-chain, no mainnet funds were at risk during the experiment.

MEV Results. Table VIII presents the MEV opportunities discovered by `MeVisor` over 90 days of Ethereum mainnet data. `MeVisor` successfully exploited 229 arbitrage and 3,468 sandwich opportunities, with an average execution time of 10.05 seconds per attack. In total, `MeVisor` generated a net profit of 426.43 ETH (USD 1,123,642.49), averaging USD 0.1445 in profit per second. In arbitrage scenarios, `MeVisor` achieved 336.40 ETH (USD 886,409.48) in revenue while incurring a gas cost of 0.87 ETH (USD 2,292.83), resulting in a net profit of 335.53 ETH (USD 884,116.65). On average, each arbitrage opportunity yielded 1.47 ETH (USD 3870.78) in profit, with a gas cost of 0.0038 ETH (USD 10.0124). For sandwich attacks, `MeVisor` identified 103.60 ETH (USD 272,975.41) in revenue and incurred total gas costs of 12.70 ETH (USD 33,449.57), resulting in a net profit of 90.90 ETH (USD 239,525.84). This corresponds to an average profit of 0.026 ETH (USD 69.07) per bundle, based on an average revenue of 0.030 ETH (USD 78.71) and a gas cost of 0.004 ETH (USD 9.65). The similar experiment to Lanturn in Appendix C shows `MeVisor` is more effective than Lanturn. These results demonstrate `MeVisor`'s effectiveness in uncovering novel, real-world MEV opportunities.

Bundle Length. `MeVisor` uncovers longer and more complex MEV opportunities that are often missed by other attackers. We focus here on arbitrage results, excluding sandwich data, since arbitrage bundle lengths vary, whereas sandwich bundles typically consist of three transactions. Table IX presents the distribution of arbitrage MEV opportunities by bundle length. `MeVisor` discovers MEV bundles ranging from 2 to 14 transactions. Notably, bundles longer than two transactions contribute a total profit of USD 798,156.12. Among these, 3 bundles reach the maximum length of 14 transactions, averaging USD 350.00 in profit, with the highest yielding 0.3078 ETH (USD 811.06).

We also observe that longer bundles do not necessarily yield higher profits, as increased complexity often results in greater gas costs and potential losses during multi-DEX swaps. Although we found 6,618 arbitrage bundles with long swaps that yielded positive revenue, 6,389 of them have insufficient profit to cover the gas cost. The majority of successful arbitrage bundles contain only two swaps. This finding aligns with real-world MEV patterns, in which most arbitrage opportunities arise between two DEXs. Therefore, `MeVisor` is effective not only in exploiting common two-transaction arbitrage bundles but also in uncovering rare, structurally complex opportunities.

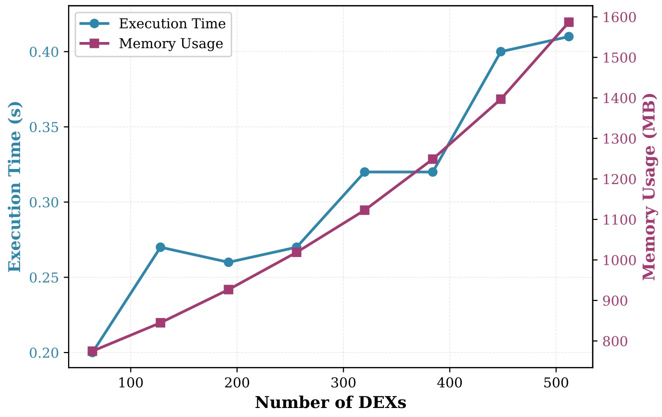


Fig. 10: Performance overhead by the number of DEXs

Overhead of `MeVisor`. On average, `MeVisor` needs 0.30s and 1.5GB memory to load these DEXs. Before searching, `MeVisor` must initialize the GPU environment and load the PTX bot, which incurs a cold boost delay of approximately 7.03 s. This cost is amortized across all subsequent searches and thus excluded from runtime overhead. During real-time searching, the primary overhead arises from loading DEX states into GPU memory. As shown in Figure 10, the loading latency remains stable at around 0.30 s even as the number of DEXs increases, whereas GPU memory usage grows proportionally from 774.88 MB (64 DEXs) to 1,586.72 MB (540 DEXs). Overall, the runtime overhead remains lightweight and the memory cost is within the capabilities of commodity GPUs, demonstrating that `MeVisor` scales efficiently with market complexity.

Answer to RQ3: `MeVisor` identifies USD 1.1 million in profit and uncovers complex MEV opportunities involving up to 14 transactions.

D. Limitations

Our study has several limitations. First, the current evaluation focuses on three major DEX protocols: Uniswap V2, Uniswap V3, and Sushiswap, which together account for most of DEX activities on Ethereum. `MeVisor` focuses on the DEXs under these DEX protocols. Since `MeVisor` is language-agnostic, it can be extended to support additional Solidity-based DEXs, such as Balancer and PancakeSwap, as well as those implemented in other languages, such as Curve.Fi. Second, we evaluate only two representative MEV strategies: arbitrage and sandwich. Another widely studied strategy is liquidation [10], [27], [55], which involves liquidating undercollateralized loans at lending protocols. We exclude this strategy because it depends on external price monitoring rather than on-chain search techniques. Nevertheless, BOTSMITH enables developers to implement new MEV strategies using Solidity interfaces. Third, we use `vm.apply` to simulate the whale transaction in sandwich attacks. It may miss relevant state changes and introduce minor inconsistencies. This limitation could be addressed by extending `vm.apply` to incorporate the full execution trace of the whole transaction, allowing complete reconstruction of affected contract states beyond DEXs. Fourth, potential inaccuracies in PTX bot simulations may cause inconsistencies when executing Uniswap V3 on GPUs. In particular, the weaker slippage control resulting from the omission of tick spacing may underestimate output values. However, this does not lead to false positives, as all solutions are eventually validated on the CPU. In future work, we plan to fully support Uniswap V3's tick spacing mechanism to enhance execution fidelity and reduce false negatives. Fifth, our large-scale analysis (§ V-C) estimates an upper bound on MEV profit because it simulates bundles on a local fork without competition. In mainnet, MEV searchers must compete with one another and may incur higher fees, such as increased gas price bidding for priority inclusion or cancellation fees from unsuccessful bundles. Future work will refine these profit estimates by simulating bundles through real-world relays [29], [59]. Relays allow simulation under realistic transaction pools and auction conditions without affecting the mainnet [28].

VI. Related Work

A. Fast EVM

The native EVM implementation [24] is primarily designed for peer synchronization and is not optimized for high-throughput execution. Several efforts have aimed to accelerate EVM performance. *Revm* [7], a high-performance EVM implementation in Rust, is widely adopted by tooling ecosystems [32], execution clients [20], [52], and emerging zero-knowledge virtual machines [44]. Other approaches improve EVM throughput by enabling parallel execution on multi-core CPUs [26], [42], [54]. To overcome the hardware limitations of CPUs, researchers have also investigated GPU-based solutions. CuEVM [37] rewrites the EVM in CUDA [48], enabling concurrent execution of multiple smart contracts on GPUs. In contrast, binary translation approaches such as MAU [9] compile EVM bytecode into SIMD programs that run natively on GPUs. MAU eliminates the EVM interpreter entirely and achieves higher throughput. However, it assumes that all cross-contract calls succeed and translates them into dummy calls, which may introduce semantic inconsistencies compared to actual on-chain behavior. In this paper, `MeVisor` introduces accurate support for cross-contract calls, enabling parallel execution of DEX interactions while preserving EVM semantics.

B. MEV Market

Since the concept of MEV was first introduced by Daian et al. [12] in 2020, many empirical studies have revealed the substantial profit potential in the MEV market and highlighted diverse transaction patterns [27], [41], [45]. Researchers have developed various methods to detect MEV opportunities Bartoletti et al. [6] proposed a theoretical model for profit maximization in DEXs. Babel et al. [4] introduced a formal verification approach to detect MEV by modeling smart contract logic. Qin et al. [56] focused on optimizing capital allocation for fixed transaction sequences, and Zhou et al. [64] applied symbolic execution to identify arbitrage cycles. Beyond optimization techniques, artificial intelligence has been increasingly applied to the MEV market [36]. For example, Babel et al. [5] used machine learning to infer MEV strategies from historical transactions and construct new MEV exploits. Most recently, Hunt et al. [63] explored the use of large language models to detect MEV opportunities in on-chain environments. However, low throughput remains a common limitation across existing tools, restricting their ability to detect MEV within the narrow inter-block interval.

C. Genetic Algorithm

Genetic Algorithms (GAs), introduced by Holland [38], are a class of metaheuristic optimization algorithms inspired by biological evolution. They evolve a population of candidate solutions through four primary operations: selection, crossover, mutation, and evaluation. Through iterative application of these operations, GAs can effectively explore large, non-differentiable solution spaces to discover high-quality solutions, such as MEV bundles. To improve scalability and performance, Alba et al. [3] proposed a parallel GPU-based implementation, leveraging the algorithm's inherent parallelism to accelerate convergence. In this paper, each individual represents a candidate MEV bundle, which is evaluated on the EVM to determine its profit. The GA's ability to operate without gradient information makes it particularly suitable for discrete optimization problems such as MEV extraction.

VII. Conclusion

We presented `MeVisor`, an efficient searcher that identifies MEV opportunities across DEXs using GPU acceleration. Extensive experiments demonstrate that `MeVisor` achieves a throughput of 3.3M-5.1M txs/sec across various MEV tasks, outperforming the state-of-the-art tool by 100,000x. This high throughput enables `MeVisor` to uncover a total of USD 1.1 million in MEV profit during Q1 2025, corresponding to a profit rate of USD 0.1445 per second.

VIII. Ethics Considerations

This GPU-acceleration technique could incentivize MEV attacks. We have shared our findings with the developers of Ethereum. Here we discuss mitigation strategies to ease MEV attacks. Protocol-level techniques such as PBS, introduced in Ethereum's MEV-Boost design [21], reduce concentrated ordering power by delegating block construction to specialized builders. Encrypted mempools, explored in systems like SUAVE [31] and Shutter Network [16], reduce pre-trade transparency to prevent opportunistic reordering. At the application layer, defences such as batch auctions [8], [46], [58], and private order-flow relays [30] mitigate common forms of retail-facing MEV in decentralized exchanges. Although these approaches cannot eliminate MEV entirely, they collectively move the ecosystem toward more transparent and equitable extraction.

Acknowledgment

We thank the anonymous reviewers and shepherds for their valuable comments and instructions. This work is supported by Hong Kong RGC Project (PolyU15231223).

References

[1] H. Adams, N. Zinsmeister, M. Salem, R. River, and D. Robinson. Uniswap v3 core. *Tech. rep., Uniswap, Tech. Rep.*, 2021.

[2] H. Adams, N. Zinsmeister, M. Salem, and D. Robinson. Uniswap v2 core. *Tech. rep., Uniswap, Tech. Rep.*, 2021.

[3] E. Alba and M. Tomassini. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, 2002.

[4] K. Babel, P. Daian, M. Kelkar, and A. Juels. Clockwork finance: Automated analysis of economic security in smart contracts. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2499–2516. IEEE, 2023.

[5] K. Babel, M. Javaheripi, Y. Ji, M. Kelkar, F. Koushanfar, and A. Juels. Lanturn: Measuring economic security of smart contracts through adaptive learning. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1212–1226, 2023.

[6] M. Bartoletti, J. Chiang, and A. Lluch. Maximizing extractable value from automated market makers. In *International Conference on Financial Cryptography and Data Security*, pages 3–19. Springer, 2022.

[7] Bluealloy. Rust implementation of the ethereum virtual machine., May. 2025. [Online]. Available: https://github.com/bluealloy/revm.

[8] E. Budish, P. Cramton, and J. Shim. The high-frequency trading arms race: Frequent batch auctions as a market design response. *The Quarterly Journal of Economics*, 130(4):1547–1621, 2015.

[9] W. Chen, X. Luo, H. Cai, and H. Wang. Towards smart contract fuzzing on gpus. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2255–2272. IEEE, 2024.

[10] F. Christof, C. Ramiro, and S. Radu. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1343–1359. USENIX Association, August 2021.

[11] Curve.Fi. Vyper contracts used in curve.fi exchange pools., May. 2025. [Online]. Available: https://github.com/curvefi/curve-contract.

[12] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE symposium on security and privacy (SP)*, pages 910–927. IEEE, 2020.

[13] W. Davidson. and S. Jinturkar. Memory access coalescing: a technique for eliminating redundant memory accesses. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 186–195, New York, NY, USA, 1994. Association for Computing Machinery.

[14] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[15] T. Diamandis, M. Resnick, T. Chitra, and G. Angeris. An efficient algorithm for optimal routing through constant function market makers. In *International Conference on Financial Cryptography and Data Security*, pages 128–145. Springer, 2023.

[16] S. Dziembowski, S. Faust, and J. Luhn. Shutter network: Private transactions from threshold cryptography. Cryptology ePrint Archive, Paper 2024/1981, 2024.

[17] F. D'Amato, B. Monnot, M. Neuder, Potuz, , and T. Tsao. Eip-7732: Enshrined proposer-builder separation [draft]," ethereum improvement proposals., May. 2024. [Online]. Available: https://eips.ethereum.org/EIPS/eip-7732..

[18] Eigenphi. Eigenphi, May. 2025. [Online]. Available: https://eigenphi.io/.

[19] E. Erdil and D. Schneider-Joseph. Data movement limits to frontier model training. *arXiv preprint arXiv:2411.01137*, 2024.

[20] Erigontech. Ethereum implementation on the efficiency frontier., May. 2025. [Online]. Available: https://github.com/erigontech/erigon.

[21] Ethereum. Proposer builder separation (pbs) ethereum, May. 2024. [Online]. Available: https://ethereum.org/en/roadmap/pbs/.

[22] Ethereum. Web3.py is a python library for interacting with ethereum., May. 2024. [Online]. Available: https://web3py.readthedocs.io/.

[23] Ethereum. The ethereum blockchain explorer., May. 2025. [Online]. Available: https://etherscan.io/.

[24] Ethereum. Golang execution layer implementation of the ethereum protocol., May. 2025. [Online]. Available: https://github.com/ethereum/go-ethereum.

[25] Ethereum. An high-level language for implementing smart contracts., May. 2025. [Online]. Available: https://docs.soliditylang.org/.

[26] Y. Fang, Z. Zhou, S. Dai, J. Yang, H. Zhang, and Y. Lu. Pavm: A parallel virtual machine for smart contract execution and validation. *IEEE Transactions on Parallel and Distributed Systems*, 35(1):186–202, 2024.

[27] C. Ferreir, A. Mamuti, B. Weintraub, C. Nita-Rotaru, and S. Shinde. Rolling in the shadows: Analyzing the extraction of mev across layer-2 rollups. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 2591–2605, 2024.

[28] Flashbot. Flashbot apis to simulate mev bundles, May. 2025. [Online]. Available: https://docs.flashbots.net/flashbots-auction/advanced/rpc-endpoint#eth_callbundle.

[29] Flashbot. Flashbots document., May. 2025. [Online]. Available: https://docs.flashbots.net/.

[30] Flashbots. Flashbots protect: Private transaction relay, May. 2024. [Online]. Available: https://docs.flashbots.net/flashbots-protect.

[31] Flashbots. The future of mev is suave, May. 2024. [Online]. Available: https://writings.flashbots.net/the-future-of-mev-is-suave.

[32] Foundry. Foundry is a blazing fast, portable and modular toolkit for ethereum application development written in rust., May. 2024. [Online]. Available: https://github.com/foundry-rs/foundry.

[33] Geckoterminal. Top ethereum pools trending today, geckoterminal, May. 2025. [Online]. Available: https://www.geckoterminal.com/eth/pools.

[34] Google. american fuzzy lop - a security-oriented fuzzer., May. 2025. [Online]. Available: https://github.com/google/AFL.

[35] Hardhat. Ethereum development environment for professionals., May. 2024. [Online]. Available: https://hardhat.org/docs.

[36] B. Henrique, V. Sobreiro, and H. Kimura. Literature review: Machine learning techniques applied to financial market prediction. *Expert Systems with Applications*, 124:226–251, 2019.

[37] Nhut-Minh Ho Ho, Ciocirlan Stefan-Dan, and Chen Li. Cuda implementation of ethereum virtual machine., May. 2025. [Online]. Available: https://github.com/sbip-sg/CuEVM.

[38] J. Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[39] M. Kalinin, D. Ryan, and V. Buterin. Upgrade ethereum consensus to proof-of-stake., May. 2025. [Online]. Available: https://eips.ethereum.org/EIPS/eip-3675.

[40] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[41] Z. Li, J. Li, Z. He, X. Luo, T. Wang, X. Ni, W. Yang, X. Chen, and T. Chen. Demystifying defi mev activities in flashbots bundle. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 165–179, 2023.

[42] H. Lin, H. Feng, Y. Zhou, and L. Wu. Parallelevm: Operation-level concurrent transaction execution for evm-compatible blockchains. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 211–225, New York, NY, USA, 2025. Association for Computing Machinery.

[43] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

TABLE X: Validation results of the motivating examples.

Length	Block	Capitial/ETH	Revenue/ETH	DEX Chain	Asset Chain
2	21536202	0.05	0.0002	0x1DC698b3d2995aFB66F96e1B19941f990A6b2662, 0x9081B50BaD8bEefaC48CC616694C26B027c559bb	0x4c11249814f11b9346808179Cf06e71ac328c1b5, 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
3	14079213	0.12	0.0009	c926990039045611eb1de520c1e249fd0d20a8ea, 62ccb80f72cc5c975c5bc7fb4433d3c336ce5ceb, 77bd0e7ec0de000eea4ec88d51f57f1780e0dfb2	0x557B933a7C2c45672B610F8954A3deB39a51A8Ca, 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2, 0xe53EC727dbDEB9E2d5456c3be40cFF031AB40A55
4	12195667	0.64	0.0196	0x1f44e67eb4b8438efe62847affb5b8e528e3f465, 0x41ca2d9cf874af557b0d75fa9c78f0131c7f345c, 0x088ee5007c98a9677165d78dd2109ae4a3d04d0c, 0xa478c2975ab1ea89e8196811f51a7b7ade33eb11	0x6b175474e89094c44da98b954eedeac495271d0f, 0xbcda9e0658f4eecf56a0bd099e6dbc0c91f6a8c2, 0x0bc529c00c6401aef6d220be8c6ea1667f6ad93e, 0x6b175474e89094c44da98b954eedeac495271d0f
5	13734406	2.04	0.0452	0x60594a405d53811d3bc4766596efd80fd545a270, 0x1d42064fc4beb5f8aaf85f4617ae8b3b5b8bd801, 0x9f178e86e42ddf2379cb3d2acf9ed67a1ed2550a, 0xfad57d2039c21811c8f2b5d5b65308aa99d31559, 0x5777d92f208679db4b9778590fa3cab3ac9e2168	0x6b175474e89094c44da98b954eedeac495271d0f, 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2, 0x1f9840a85d5af5bf1d1762f925bdaddc4201f984, 0x514910771af9ca656af840dff83e8264ecf986ca, 0x6b175474e89094c44da98b954eedeac495271d0f
6	14530401	0.41	0.0182	0xa478c2975ab1ea89e8196811f51a7b7ade33eb11, 0x5ab53ee1d50eef2c1dd3d5402789cd27bb52c1bb, 0x59c38b6775ded821f010dbd30ecabdcf84e04756, 0x9f178e86e42ddf2379cb3d2acf9ed67a1ed2550a, 0x153b4c29e692faf10255fe435e290e9cfb2351b5, 0x1f44e67eb4b8438efe62847affb5b8e528e3f465	0x6b175474e89094c44da98b954eedeac495271d0f, 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2, 0x7fc66500c84a76ad7e9c93437bfc5ac33e2ddae9, 0x1f9840a85d5af5bf1d1762f925bdaddc4201f984, 0x514910771af9ca656af840dff83e8264ecf986ca, 0x6b175474e89094c44da98b954eedeac495271d0f
7	13731798	0.24	0.0202	0x1f44e67eb4b8438efe62847affb5b8e528e3f465, 0x153b4c29e692faf10255fe435e290e9cfb2351b5, 0x14243ea6bb3d64c8d54a1f47b077e23394d6528a, 0xd75ea151a61d06868e31f8988d28dfe5e9df57b4, 0x088ee5007c98a9677165d78dd2109ae4a3d04d0c, 0x41ca2d9cf874af557b0d75fa9c78f0131c7f345c, 0x1f44e67eb4b8438efe62847affb5b8e528e3f465	0x6b175474e89094c44da98b954eedeac495271d0f, 0xbcda9e0658f4eecf56a0bd099e6dbc0c91f6a8c2, 0x514910771af9ca656af840dff83e8264ecf986ca, 0x7fc66500c84a76ad7e9c93437bfc5ac33e2ddae9, 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2, 0x0bc529c00c6401aef6d220be8c6ea1667f6ad93e, 0x6b175474e89094c44da98b954eedeac495271d0f
8	13754757	2.48	0.0680	0xa478c2975ab1ea89e8196811f51a7b7ade33eb11, 0x5ab53ee1d50eef2c1dd3d5402789cd27bb52c1bb, 0x59c38b6775ded821f010dbd30ecabdcf84e04756, 0x1d42064fc4beb5f8aaf85f4617ae8b3b5b8bd801, 0x11b815efb8f581194ae79006d24e0d814b7697f6, 0xfcd13ea0b906f2f87229650b8d93a51b2e839ebd, 0xc0067d751fb1172dbab1fa003efe214ee8f419b6, 0x60594a405d53811d3bc4766596efd80fd545a270	0x6b175474e89094c44da98b954eedeac495271d0f, 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2, 0x7fc66500c84a76ad7e9c93437bfc5ac33e2ddae9, 0x1f9840a85d5af5bf1d1762f925bdaddc4201f984, 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2, 0xdac17f958d2ee523a2206206994597c13d831ec7, 0x4206931337dc273a630d328da6441786bfad668f, 0x6b175474e89094c44da98b954eedeac495271d0f

[44] Matter-Labs. A high-performance rewrite of the out-of-circuit vm for zksync era (aka eravm)., May. 2025. [Online]. Available: https://github.com/matter-labs/vm2.

[45] R. McLaughlin, C. Kruegel, and . Vigna G. A large scale study of the ethereum arbitrage ecosystem. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3295–3312, Anaheim, CA, August 2023. USENIX Association.

[46] C. McMenamin, V. Daza, M. Fitzi, and P. O'Donoghue. Fairtradex: A decentralised exchange preventing value extraction. In *Proceedings of the 2022 ACM CCS Workshop on Decentralized Finance and Security*, pages 39–46, 2022.

[47] E. Miller and E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[48] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, March 2008.

[49] Nvidia. Nvvm-specific intrinsics for use with nvptx, May. 2025. [Online]. Available: https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/IR/IntrinsicsNVVM.td.

[50] Nvidia. Parallel thread execution isa, May. 2025. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/.

[51] Paradigm. Estimates the eip1559 maxfeepergas and maxpriorityfeepergas fields., May. 2025. [Online]. Available: https://docs.rs/alloy-provider/latest/alloy_provider/trait.Provider.html#method.estimate_eip1559_fees.

[52] Paradigmxyz. Blazing-fast implementation of the ethereum protocol., May. 2025. [Online]. Available: https://github.com/paradigmxyz/reth.

[53] Pybind. Seamless operability between c++ and python., May. 2025. [Online]. Available: https://github.com/pybind/pybind11.

[54] X. Qi, X. Chen, and N. Han. Boosting blockchain throughput: Parallel evm execution with asynchronous storage for reddio, 2025.

[55] K. Qin, L. Zhou, and A. Gervais. Quantifying blockchain extractable value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 198–214. IEEE, 2022.

[56] K. Qin, L. Zhou, B. Livshits, and A. Gervais. Attacking the defi ecosystem with flash loans for fun and profit. In *International conference on financial cryptography and data security*, pages 3–32. Springer, 2021.

[57] Sushiswap. Sushiswap document., May. 2025. [Online]. Available: https://docs.sushi.com/what-is-sushi.

[58] CowSwap Team. Cowswap: A batch-auction dex for mev protection, 2025.

[59] Titan. Titan builder document., May. 2025. [Online]. Available: https://docs.titanbuilder.xyz/api.

[60] Vyper. A contract-oriented, pythonic programming language that targets the evm., May. 2025. [Online]. Available: https://vyper.readthedocs.io/.

[61] B. Weintraub, C. Ferreira, C. Nita-Rotaru, and R. State. A flash(bot) in the pan: Measuring maximal extractable value in private pools. In *Proceedings of the 22nd ACM Internet Measurement Conference (IMC '22)*, Nice, France, 2022. Association for Computing Machinery.

[62] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[63] Z. Wu, J. Wu, H. Zhang, Z. Zheng, and W. Wang. Hunting in the dark forest: A pre-trained model for on-chain attack transaction detection in web3. In *Proceedings of the ACM on Web Conference 2025*, WWW '25, page 4519–4530, New York, NY, USA, 2025. Association for Computing Machinery.

[64] L. Zhou, K. Qin, A. Cully, B. Livshits, and A. Gervais. On the just-in-time discovery of profit-generating transactions in defi protocols. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 919–936. IEEE, 2021.

Appendix A
DEX Profit Example.

DEXs dynamically adjust token-pair prices based on the reserves. For example, in a WETH/USDT Uniswap pool, the product of reserves of WETH and USDT is a constant. When a victim swaps a large amount of WETH for USDT, the WETH reserve in the DEX increases and the USDT reserve decreases, making USDT more expensive relative to WETH after the trade. This price impact motivates attackers to buy USDT before and sell USDT after the victim's trade to profit WETH from the induced price change.

Appendix B
Comparison with MAU

To tackle the MEV searching problem, `MeVisor` introduces three major innovations that enable efficient, fully GPU-driven execution, compared to MAU [9]. First, `MeVisor` adds support for cross-contract calls essential to DEX execution, which MAU lacks. Specifically, we translate each smart contract into a function and therefore the EVM call site is simulated by a function call. This design allows `MeVisor` to execute complex DEX interactions across multiple contracts entirely within the GPU. Second, `MeVisor` implements efficient GPU memory management for smart contract execution. In contrast to MAU's costly per-call dynamic allocation, `MeVisor` adopts a static allocation strategy that reuses EVM components across call frames while preserving isolation, effectively removing runtime allocation overhead and improving scalability for cross-contract execution. Third, `MeVisor` incorporates a GPU-parallelized genetic algorithm to explore and evaluate bundles of transactions for potential profit. Unlike MAU, which acts as a CPU-guided fuzzer that selects input batches and delegates their execution to the GPU, `MeVisor` performs both the search and execution entirely on the GPU. This fully on-GPU feedback loop removes CPU bottlenecks and allows `MeVisor` to continuously evolve transaction bundles in real time for higher profit.

Appendix C
Lanturn Large-scale Evaluation.

We also include an evaluation with Lanturn under the same time constraints used in RQ3. Lanturn relies on predefined transaction templates that are typically crafted by MEV experts. To compare against Lanturn, we use both the default templates provided in Lanturn's codebase and additional MEV templates extracted from historical MEV incidents. As a result, Lanturn identifies 114 new arbitrage opportunities, yielding a total profit of 1.819 ETH with 1.943 ETH revenue and 0.124 ETH gas cost. As for the sandwich attacks, Lanturn identifies 4 new opportunities, yielding a total profit of 0.048 ETH with 0.051 ETH revenue and 0.003 ETH gas cost. These results indicate that `MeVisor` is more effective in uncovering on-chain MEV profits.