Light into Darkness: Demystifying Profit Strategies Throughout the MEV Bot Lifecycle

Feng Luo[†], Zihao Li[†], Wenxuan Luo[‡], Zheyuan He[‡], Xiapu Luo[†✉] ,
Zuchao Ma[†], Shuwei Song[‡] and Ting Chen[‡✉]
[†]The Hong Kong Polytechnic University
[‡]University of Electronic Science and Technology of China

***Abstract*—Due to the transparency of permissionless blockchain, opportunistic traders can extract Maximal Extractable Value by competing for profit opportunities and making the process never stop by creating MEV bots. However, this behavior undermines the consensus security and efficiency of the blockchain system. Therefore, understanding the behavior strategies of MEV bots is crucial to protect against their harm. Unfortunately, existing work focuses on macro-measurements of the MEV market, and the specific types and distributions of MEV bot strategies remain unknown. In this paper, we conduct the first systematic study of MEV bot profitability strategies by developing APOLLO, a tool to analyze fine-grained strategies throughout the entire lifecycle of bots. Our large-scale analysis of 2,052 MEV bots yields many new insights. In particular, we first introduce 20 code-level strategies employed by bots in the wild, take the first step towards smart contract de-obfuscation to discover strategies hidden in obfuscated bot code, and discover five specific types of transactions that bring profit opportunities to MEV bots.**

I. INTRODUCTION

DeFi has revolutionized the finance industry with the total locked-in value of over 79.3B USD [1]. The operation of DeFi relies on smart contracts, and transactions broadcasted on the permissionless blockchain are publicly visible on the global P2P network [2], [3], [4]. Therefore, once a user sends a profitable transaction, opportunistic traders can prioritize their transactions by increasing the transaction fee, appropriating the trading opportunity, and earning additional revenue from it, which is known as MEV [5], [6], [7], [8], [9].

Competition among MEV extractors undermines the fairness and security of the blockchain. First, competitors vie for an advantage by competitively raising transaction fees, leading to a gas war that implicates normal users [10]. Second, MEV causes users or liquidity providers to suffer unexpected asset losses, damaging blockchain economic security [9]. Furthermore, MEV competition incentivizes financially rational miners to fork the chain, thus deteriorating the blockchain consensus security [6]. Notably, MEV extractors, in pursuit of higher revenues, deploy MEV bots on the chain. These bots perpetually seek opportunities and execute MEV transactions during their lifecycle. This means that the impacts of MEV are ever-present, and the bot's high-frequency transactions exacerbate network and block congestion, causing considerable time delays for other transactions [6], [8].

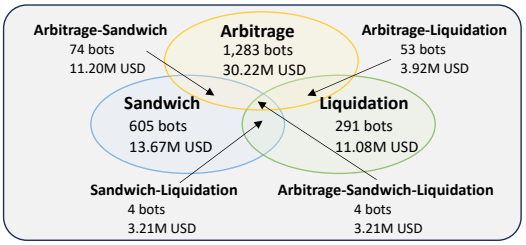

Fig. 1. MEV bot statistics

There are already a number of studies on MEV. Some researchers have dissected the rationale for different types of MEV and performed quantitative analyses to evaluate the tangible risks that MEV poses [8], [11], [12], [13], [14]. In addition, researchers have proposed methods to detect MEV behavior and contributed methods to mitigate the impact of individual MEV types, providing critical insights to the blockchain community [15], [5], [16].

Despite substantial studies on the macro MEV behavior, the detailed strategies MEV bots employ remain unknown. MEV bots use sophisticated, hard-to-detect lifecycle strategies that undermine blockchain fairness to profit. Demystifying these strategies is crucial as: ① aids developers in understanding every action of MEV strategies that could harm blockchain security in the wild, guiding for building defensive measures; ② reveals areas in the DeFi ecosystem that exploitable for profit and require enhancement; ③ helps users understand what types of transactions expose value and become targets for bots.

In this paper, we conduct the first systematic study of fine-grained strategies for MEV bots on Ethereum. We conceptualize bots' entire lifecycle and divide it into **birth-running-death** phases. We then define 11 types of strategies spanning the bot lifecycle, encompassing bot owners' diverse tactics and designs. Specifically, in the **birth** phase, bot owners can inject code-level strategies S_{code} into the bot, including embedding business logic strategies S_{logic}, using obfuscation techniques to prevent reverse analysis, and using gas optimization strategies S_{gas} to reduce operating costs, and then deploy the bot to the blockchain. In the **running** phase, the bot identifies

✉ The corresponding authors.

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.2026.230506
www.ndss-symposium.org

profitable token exchange paths by analyzing tokens and token pairs, and performs opportunity discovery by focusing on catalyst transactions that may yield opportunities. For these opportunities, the bot determines the source of initial funding and the distribution of eventual profits, forming the outline logic. It then sets specific parameters for the MEV extraction behavior by balancing risk and profit, and designing the transaction's $calldata$ and $value$ fields. The transaction is initiated and executed through mempool broadcasting or the builder service. The **death** phase concludes the lifecycle, occurring when a bot is decommissioned and ceases transaction activities.

To comprehensively analyze MEV bots, we design APOLLO, the first tool combining contract code and transactions to analyze bot's code strategies S_{code} and transaction incoming strategies S_{tx}. Implementing such a tool is non-trivial. First, we lack knowledge about S_{logic} because there is no ground truth to reveal specific strategy types. To gain knowledge, manual inspection is inevitably involved to discover unknown strategies. To reduce manual effort, we use feature extraction techniques to capture semantic features of unknown strategies and employ iterative clustering to facilitate the discovery of new strategy types. Second, due to the importance of profitability, MEV bot owners likely protect their bots from S_{logic} leakage using obfuscation techniques. The challenge of smart contract de-obfuscation is significant and has previously relied on manual analysis. To address this, we take the first step towards contract de-obfuscation. We design an EVM opcode-based Markov transition matrix and use a CNN to automatically analyze obfuscated bot bytecode.

Third, identifying catalyst transactions that bring opportunities for MEV bots is an unknown causal analysis problem. Lack of prior knowledge hinders the design of deterministic detection patterns. Therefore, we design a semi-automated prior knowledge harvesting process, which first analyzes historical transactions potentially related to MEV activities using association rule learning (ARL), and then manually analyzes the relationships between historical and MEV transactions with the assistance of ARL-generated rules. We find that large transactions, transactions involving new altcoins, same sender transactions following failed transactions, transactions lacking execution chain check, and transactions with excessive gas prices can act as MEV catalysts. Based on the knowledge, we design targeted detection algorithms for these five types of transactions to automatically detect catalyst transactions.

To unveil the strategies applied by the MEV bots running in the chain, we use APOLLO to analyze 2,052 MEV bots. As shown in Fig. 1, these bots consist of 1,283 arbitrage, 605 sandwich, and 291 liquidation bots, which made a total of 2,516,236 transactions and a total of 40M USD profit. By analyzing them, we gained a lot of novel insights. For example, 16.3% (334) of bots employ obfuscation to protect their code logic from disclosure. 24.7% (506) of bots incorporate S_{logic} that perform critical parameter calculations and strategy choices during trade execution, while 75.3% (1546) of bots only function as execution agents, relying on invocation transactions initiated by the off-chain program that contain the full behavioural logic. In addition, MEV bots can reduce the execution cost to improve profitability through methods such as gas optimisation as well as calldata compression.

In summary, this paper makes four contributions.

- *Comprehensive MEV bot analysis.* We propose APOLLO, a tool for systematically analyzing MEV bot strategies based on contracts and transactions. It reveals the strategic details employed by wild bots that threaten blockchain security.
- *Contract de-obfuscation analysis.* We propose a new method based on the opcode Markov transition matrix to analyze strategies of obfuscated bots.
- *New study on catalyst transaction.* We design an ARL-based process to analyze which transactions present profit opportunities to the MEV bot and discover five types of catalyst transactions with exploitation risks.
- *New insights.* We conduct a large-scale analysis of 2,052 MEV bots. This analysis uncovers the various bot strategies, yields many new insights, and provides relevant mitigation suggestions, enhancing the DeFi ecosystem security. We release materials of our work in https://github.com/sec-study-dev/apollo or future research.

II. BACKGROUND

A. Smart Contract and Transaction

Smart contracts are programs that define a set of execution logic and are deployed to the blockchain after being compiled into bytecode [17], [18]. Smart contracts automatically execute according to predefined logic when triggered by a transaction, and transaction fees are charged based on the executed instructions [19], [20]. Transactions pass the target call function and required parameters through the $calldata$ field, and pass the amount of native currency (e.g., ETH on Ethereum) to be transferred to the recipient through the $value$ field [21], [22].

B. Decentralized Finance

AMM. Automated Market Maker exchanges are peer-to-peer trading platforms that enable users to exchange cryptocurrencies without involving a third party. AMMs maintain liquidity pools containing at least two assets and use specific computational models to automate trades within these pools [5], [23].
Slippage. It is the discrepancy between the expected and actual price when trading in AMM. The expected price is determined based on blockchain state at the time of transaction initiation. There is a delay between transaction initiation and execution, during which slippage can occur if the related blockchain state changes due to other transactions [5]. Therefore, traders typically set slippage protection for transactions to determine the maximum acceptable slippage and avoid bad execution prices.
Lending. It allows borrowers to collateralize their deposits to borrow loanable assets [5]. Lending platforms usually require over-collateralization, where the value of the collateral exceeds the assets lent by the borrower [12]. If the value of the collateral decreases and the collateralization ratio (the value of the collateral relative to the debt) falls below a specified threshold, the platform can liquidate the collateral and sell it at a discount to repay the debt [8].

C. Maximal Extractable Value

MEV is the additional revenue that speculative traders earn from DeFi by strategically adjusting the gas price to manipulate the order of transactions [5].

Arbitrage. It profits from changes in market prices by simultaneously buying and selling assets in AMMs [13].

Sandwiches. Traders can continuously monitor the blockchain to find the victim's pending trades T_V, and then wrap T_V in two adversarial transactions T_{A1} and T_{A2}. T_{A1} frontruns T_V to buy the asset at a low price, and T_{A2} backruns T_V to sell the same asset at a higher price to profit from T_V [8].

Liquidation. It is a way for liquidators to make a profit by buying collateral below market price on a lending [12].

Proposer-Builder Separation (PBS). PBS is a scheme that emerged after Ethereum transitioned from Proof-of-Work to Proof-of-Stake, in which the roles of the proposer (usually the validator) and the builder are separated [24]. The builder is responsible for building the sequence of transactions to create block proposals and submitting them to the validator, who is responsible for submitting those blocks to the blockchain. Transaction initiators can send bundles containing their own transactions as well as other transactions from the Ethereum mempool to one or more block builders. In addition, a transaction initiator can also submit transactions directly through the mempool without going through a specific builder [24].

D. Smart Contract Obfuscation

Deployers can obfuscate contracts to protect intellectual property rights or avoid attacks due to business logic leaks. There are many works devoted to contract obfuscation [26], [25], [27], [28], and the obfuscation techniques they involve include the following categories: ① Control flow obfuscation (CFO): Alter or complicate the control flow of a program to make it more difficult to trace or reconstruct the execution flow. ② Data flow obfuscation (DFO): Modify the data domains and structures of a program, including variable storage and coding obfuscation, combining or splitting variable obfuscation, and order adjustment obfuscation. ③ Layout obfuscation (LO): Remove irrelevant information from a program or replace class and method names. ④ Code complexity (CC): Add redundant and meaningless code. Table I shows the obfuscation techniques used by existing open source tools.

III. MEV Bot and Strategies Taxonomy

MEV bots are on-chain programs that identify profitable opportunities and execute trades automatically. Existing research identifies two attack surfaces where bots discover opportunities: ① current block state s_{block} and ② mempool state $s_{mempool}$. For ①, bots directly monitor the s_{block} and discover exploitable opportunities based on token data. For ②, bots monitor the $s_{mempool}$, which allows the bot to predict future opportunities since pending transactions, once executed, change the block state and may spawn new opportunities. For an opportunity $\mathcal{O}$, the bot can identify a token exchange path $\{T_0 \xrightarrow{pair} T_1 \cdots T_{n-1} \xrightarrow{pair} T_n\}$ involving n tokens and profit through multiple strategies. The bot's gain from $\mathcal{O}$ as Eq.1.

$$P(\mathcal{O}, S_{code}, S_{tx}) = \underbrace{\sum R(T_i)E(T_i)}_{Revenue} - \underbrace{F(\mathcal{O})E(ETH)}_{Cost} \quad (1)$$

where S_{code} is a set of code-level strategies, which are logics implemented by the bot owner within the contract code and run on-chain during transaction execution. S_{tx} is a set of transaction-level strategies, which are execution logics formulated by the bot owner off-chain that are passed to and guide the bot's actions through transactions. $R(T_i)$ is the amount of change in the balance of token T_i. $E(T_i)$ is the exchange rate of token T_i to USD. F is the transaction fee cost. $E(ETH)$ is the exchange rate of Ether. We further define $\mathbb{E}[P(\mathcal{O})]/\mathbb{A}[P(\mathcal{O})]$ as the expected/actual gain from $\mathcal{O}$.

Our goal. Comprehensive analysis of the strategies S_{code} and S_{tx} adopted by the MEV bots exposes how the on-chain bots discover the opportunity $\mathcal{O}$ and how they profit from it.

To achieve this goal, we divide the entire lifecycle of MEV bots into three stages of birth, running, and death as shown in Fig. 2, and deeply analyze the strategies of each stage. MEV bots are written into S_{code} and deployed to the blockchain in the birth stage, execute MEV transactions based on S_{code} within the contract and incoming S_{tx} from transactions in the running stage, and are abandoned to stop activities in the death stage. For bots, on-chain logic is determined by the S_{code} within the contract during transaction execution, so it can be obtained by analyzing the bot contract. Off-chain logic is not in the contract and is defined off-chain and passed in via S_{tx}, so it can be obtained by analyzing transaction behavior.

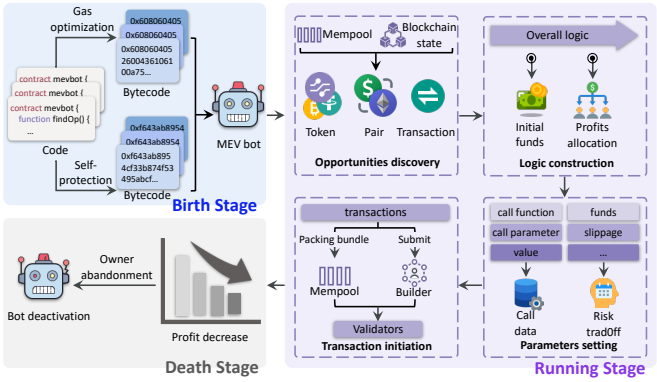

Fig. 2. The lifecycle of MEV bots.

A. Birth Stage

Contracts, once deployed to the blockchain, are immutable, prompting deployers to inject all code-level strategies S_{code} into the bot code before deployment. S_{code} is of two primary types: business logic strategies S_{logic} related to MEV extraction and gas optimisation strategies S_{gas} to reduce cost.

TABLE I
Existing Obfuscation Tools

Tool	Techniques	Object
BiAn [25]	CFO, DFO, LO	Source code
BOSC [26]	CFO, DFO, CC	Bytecode
Eshield [27]	CFO	Bytecode
Evmcodegen [28]	CFO, DFO, CC	Bytecode

❶ **S1: Business logic strategies.** S_{logic} exists in bot contract bytecode, and anyone can access it due to the blockchain's transparency. Attackers can analyze the bot's S_{logic} from the bytecode and perform targeted attacks, causing huge damage to the bot. On April 3, 2023, multiple MEV bots were attacked and lost more than $25M [29]. Therefore, cautious developers use contract obfuscation to prevent S_{logic} leaks [26], [25].

❷ **S2: Gas optimization.** As shown in Eq. 1, the cost of MEV bots comes from the transaction fee, which is impacted by the instructions involved in the contract execution. Therefore, optimizing gas directly enhances the bot's profitability by replacing the original instruction sequence with an anti-sequence that performs the same function but requires less gas.

B. Running Stage

This stage is the main working stage of MEV bots, which executes transactions according to established logic. We further divide the stage into four sub-stages: opportunities discovery, logic construction, parameters setting, and transaction execution, for a more detailed analysis based on different periods of the bot's workflow.

1) Opportunities discovery: MEV bots monitor block state and mempool state to identify exploitable opportunities. We first define the concept of catalyst transaction.

*Definition 1 (**Catalyst Transaction**):* A transaction $\mathcal{T}$ is a catalyst transaction for MEV bots if its execution can create opportunity $\mathcal{O}$, and based on the tokens affected by $\mathcal{T}$ can find the suitable strategies $S_{code} \cup S_{tx}$ such that the opportunity $\mathcal{O}$ has an expected profit $\mathbb{E}[P(\mathcal{O})] > 0$.

Due to the transparency of the blockchain, MEV bots can identify pending transactions predicted to be catalyst transactions from the mempool [30] and perform the corresponding token exchange operation. Thus, we analyze the opportunity discovery strategy from the following two perspectives.

❸ **S3: Identify profitable exchange paths.** An exchange path consists of tokens and token pairs. All tokens and associated pairs form a complex network, with multiple paths between any two tokens. MEV bots need to identify the appropriate combination of tokens to maximize profits.

❹ **S4: Search for catalyst transactions** If bots identify a catalyst $\mathcal{T}$ that brings an opportunity, they can predict an exchange path faster based on tokens associated with $\mathcal{T}$.

2) Logic Construction: After determining the token path, MEV bots perfect the overall execution logic. This logic outlines the expected profit process, including the investment of initial funds, the allocation of profits in the final phase.

❺ **S5: Initial funds.** Initial funding is the token investment required for MEV bot to perform MEV transactions. In blockchain, there are two sources of initial funding. First, it is provided by the bot owner. Second, it is borrowed through flashloan and returned after the transaction is completed.

❻ **S6: Profits allocation.** Depending on the role, profit destinations in MEV transactions can be categorized into five types: MEV bot, transaction sender, builder, validator, and other addresses. Other addresses may include those used by the bot owner to hold funds and those of professional MEV bot code providers, who offer ready-to-use code to the public, enabling participants without expertise to extract MEV and take a proportional cut of the profits.

3) Parameters Setting: After the overall logic, the action details can be clarified by injecting specific parameters.

❼ **S7: Risk tradeoff.** MEV extraction is a prospective behavior, as the reserves of all tokens, real-time exchange rates, and transaction formulas in AMMs are publicly visible. Additionally, the transaction fee for executing an MEV transaction can be estimated in advance using the `eth_estimateGas` RPC method. Thus, for an opportunity $\mathcal{O}$, the MEV bot can accurately calculate the theoretically optimal source token investment OT_0 that will yield the maximum expected profit $max(\mathbb{E}[P(\mathcal{O})])$ in the current state. However, the MEV bot may not select the optimal investment OT_0 due to risk management. For example, in a sandwich transaction, OT_0 might cause the victim transaction V_m to trigger slippage protection due to excessive losses, leading to the failure of the transaction. Therefore, the actual input RT_0 of the MEV bot may differ from OT_0. We refer to the difference between RT_0 and OT_0 as the bot's trade-off between profit and risk, reflecting the MEV bot's consideration of transaction success.

❽ **S8: Calldata.** A transaction's $calldata$ contains the parameters passed to the bot by the transaction initiator, and $calldata$ length affects the transaction fee. Specifically, processing each non-zero byte of $calldata$ requires 16 gas, and each zero byte requires 4 gas [31]. Therefore, professional developers compress $calldata$ to reduce costs. For example, each parameter in Ethereum is passed in a 32-byte format, and parameters less than 32 bytes are padded with zeros. Therefore, $calldata$ length can be reduced by eliminating redundant zeros.

❾ **S9: Value.** The $value$ field in transactions is commonly used to transfer ETH. However, we discover that some bots use $value$ to pass parameters instead of transferring ETH. By carefully designing $value$ and reading it as function parameters at execution, the $calldata$ can be further compressed.

4) Transactions Execution: Based on the overall logic and specific parameters, MEV bots submit the transaction in different ways to extract MEV.

❿ **S10: Builder distribution.** For MEV bots, submitting transactions via the mempool provides more autonomy and avoids sharing profits with the builder. Submitting transactions via the builder reduces the risk of malicious attacks because transactions are not publicly exposed in the mempool. Therefore, how to submit transactions is an important strategy.

C. Death Stage

Not all MEV bots are continuously active, and many are currently inactive on the chain. For an MEV bot, the last strategy that the owner makes for them is to discard them under what circumstances.

⓫ **S11: Death.** Given that MEV bots are profit-driven, it is reasonable to assume that the reason for their abandonment is a decline in profitability.

IV. APOLLO

A. Overview

As shown in Fig. 3, APOLLO inputs the transactions and bytecodes of an MEV bot and then analyzes its strategies (**S1-11**) across three stages. Notably, some strategies we aim to analyze depend on real-time data, such as instantaneous token prices and reserves. This prevents us from directly using historical token data provided by data services as in prior MEV measurement work [8], [15], as these services only provide data for specific dates or blocks and not for any time point. Therefore, we design algorithms that reproduce historical token data at any given moment, allowing us to replicate the conditions MEV bots faced in the past. Appendix A show the detailed process and evaluation of the algorithm. The results show that our algorithm can provide more detailed and accurate historical data than existing data sources.

B. Birth Stage Strategy Analysis

❶ **S1: Business logic strategies.** Since no current work analyzes MEV bot business logic strategies S_{logic}, there is a lack of ground truth that reveals the specific type of S_{logic}. To escape this dilemma, as shown in Fig. 3, for the input bot bytecode, APOLLO first determines whether it is code-obfuscated. Iterative learning is performed on unobfuscated bots to mine refined strategy semantics and complement the strategy knowledge in the process. Further, APOLLO analyzes the obfuscated bots based on acquired knowledge.

• **Obfuscation judgement.** To determine whether bot is obfuscated, APOLLO first checks whether the inherent structure of its bytecode has been modified, then analyzes whether control flow and data flow are obfuscated. Detailed judgment steps can be found in Appendix B.

• **Unobfuscated bot analysis.** APOLLO extracts the instruction sequences related to profit strategies and embeds the strategy semantics into the vector space via word embedding. These embeddings will be used to perform natural clustering based on distinct strategy semantics through the clustering algorithm, facilitating unknown strategy type identification.

- Bytecode slicing. Contracts usually contain many instructions that are not related to profit strategies. These noises hinder the model to understand the S_{logic} semantics. To minimize noise interference, APOLLO slices the bytecode by taint analysis, keeping only the instructions that reflect the S_{logic} semantics.

Specifically, APOLLO first simulates bytecode execution to track the control and data flow [32]. To identify instructions related to the profit strategy, APOLLO locates instructions that indicate the token transfer behavior and identifies the transfer amount and address involved as tainted data sources. In more detail, there are two types of transfers in a contract: native token transfers (e.g., ETH) and non-native token transfers (e.g., various tokens that follow the ERC protocol). For native token transfers, it executes a `CALL` or `CALLCODE` instruction to get the transfer amount, sender, and receiver from the stack. For non-native token transfers, it uses the `LOG` instruction to record the transfer event's amount, sender, and receiver. We use the amount, sender, and receiver as the taint source and perform a backward taint analysis. Tracing the arguments and outputs of all instructions involved in taint propagation to identify instructions related to profit computation and distribution strategies. For example, Fig. 4 illustrates a slicing process for logic related to amount computation.

- Feature extraction. To enable the model to distinguish the semantics of different unknown strategies, semantic features represented by the instruction sequences in slices need to be embedded into the vectors. For this, APOLLO uses representation learning, a method of automatically mining the intrinsic features of the data, to perform feature capture. APOLLO uses spatial pattern models, rather than time series models (e.g., LSTM, Transformer) for learning. Because the instructions in a slice are arranged in the order of execution, the inherent temporal properties of these instructions have been characterized as spatial sequential patterns. Spatial models outperform time series models for such data by capturing both spatial and temporal information [33]. Among spatial models, APOLLO uses CNN because it demonstrates superior performance over other models with limited datasets [34]. Given the limited number of bots present in the wild, CNN was chosen to ensure APOLLO's performance under these constraints.

As shown in Fig. 3, the feature extraction consists of an embedding, a CNN, and a fully connected (FC) layer. For input slices, APOLLO uses skip-gram [35] embeddings to encode the contextual relevance of the instructions in each slice and transform them into initialization vectors. Given each initial embedding, the CNN layer refines the embedding matrix by stacking the multi-view convolutional and the max pooling layer to aggregate instruction information. This network structure facilitates feature extraction as: 1) The convolution layer that adaptively generates filters of different kernel sizes can extract the interactions between instructions in different windows to enrich the captured semantic features. Each filter forms a new embedding matrix. Concatenating these new embedding matrices enables the characterization of more fine-grained strategy semantics. 2) The max pooling layer aggregates local features to highlight the most important ones.

To facilitate classification, the FC layer performs dimensionality reduction to output low-dimensional vectors. APOLLO builds multi-label classifiers based on multilayer perceptron (MLP) [36] and trains the model to perform classification. The initial label used for classification is "unknown", and APOLLO assigns initial labels to unrecognizable samples. After discovering new strategy types in each round of clustering analysis, we extend the new strategy labels to the classifier to guide it to perform more accurate classification.

- Iterative cluster learning. For **S1**, APOLLO is analyzed semi-automatic iteratively. The clustering aims to create effective groupings for code slices by clustering the corresponding vectors. Focusing on each group minimizes the manual work required to identify new strategy types and labels. For the clustering model, we tested four candidates, Genieclust [37], DBSCAN [38], HDBSCAN [39], and K-Means [40]. Genieclust was ultimately chosen for two reasons: 1) Unlike K-Means,

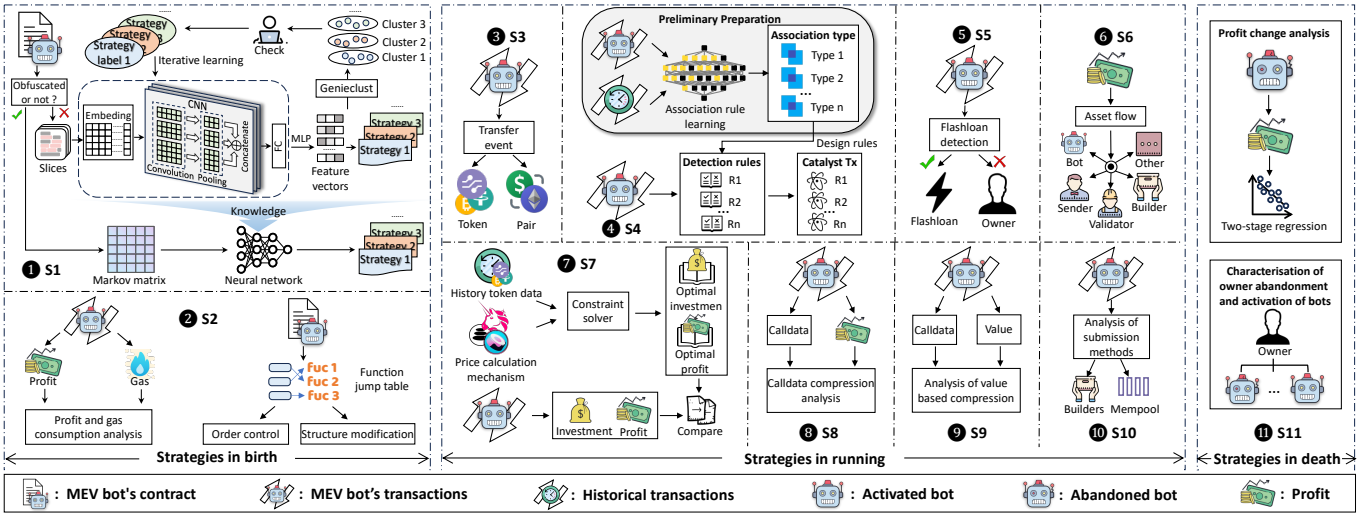

Fig. 3. Overview of APOLLO.

Genieclust does not require a fixed number of clusters to be pre-specified for initialization. This makes APOLLO more robust in scenarios lacking knowledge about the type and number of strategies. 2) Genieclust is more robust against noisy data points compared to DBSCAN and HDBSCAN [41]. This can help APOLLO distinguish semantic features more accurately to discover unknown strategies. During manual validation, once new strategy are discovered, the new labels are incorporated into the CNN model, allowing it to iteratively optimize the feature learning process for more accurate classification. In each iteration, we manually analyze only the denser clusters (i.e., more than 50% of the samples with similar patterns), while the rest of the clusters move on to the next iteration. This approach reduces unnecessary analysis costs, especially in the early running stages of APOLLO, where insufficient strategy knowledge may lead to inaccurate clustering.

• **Obfuscated bot analysis.** There is no current research analyzing obfuscated contracts and the failure rate of existing methods dealing with obfuscated contracts is close to 100% [26]. Therefore, to identify strategies for obfuscated bots, new methods need to be designed. We draw inspiration from Java de-obfuscation work to address this challenge. Opcode-based Markov transition matrices are widely used in Java bytecode de-obfuscation and achieve significant results [42], [43]. Compared to Java, Solidity opcodes are fewer in number and more unambiguous in function, e.g., `MLOAD` in Solidity is used to load data from memory, whereas `LDC` in Java can load constants of various types such as numbers, strings, class and method references, etc. Therefore, applying the opcode

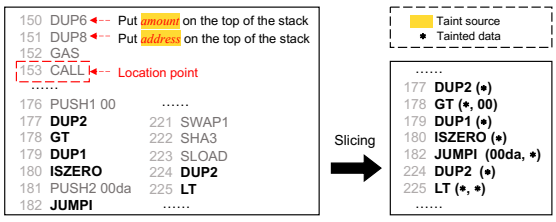

Fig. 4. The process of slicing

Markov transition matrix for contract de-obfuscation makes sense. Specifically, we design an EVM opcode-based Markov transition matrix (EOMM), an instruction-level feature that serves as a semantic feature of the contract. This semantic feature is constructed using the transfer relationships among opcodes. Fig. 5 shows the process of APOLLO analyzing the obfuscated bots. It extracts the EOMM features from the sequence of opcodes in the bytecode and then uses a CNN to learn vector representations to detect the bot's S_{logic}.

- ***EOMM feature.*** Since code obfuscation does not change the opcodes themselves, instruction-level EOMM features are inherently resistant to techniques such as variable or function name obfuscation [43]. Each opcode in Solidity corresponds to an 8-bit hexadecimal number. The opcode sequence reflects the contract's behavior at runtime [44], so APOLLO extracts the opcode sequence $(O_1, \ldots, O_i, \ldots, O_n)$ from the bytecode, where i represents the opcode number. The range of opcode numbers is from 0x00 to 0xff, i.e., 256 positions. To characterize the opcode collocation relationship, we construct a 256×256 Markov transition matrix as the EOMM as shown in Fig. 5. $P_{i,j} = \frac{q(i,j)}{\sum_{m=0}^{m=255} q(i,m)}$, denotes the probability of transiting from opcode i to opcode j, where $q(i,j)$ denotes the number of transitions from opcode i to j.

- ***Classification.*** Since CNN efficiently processes matrix data [34], we train a CNN to handle the EOMM and perform

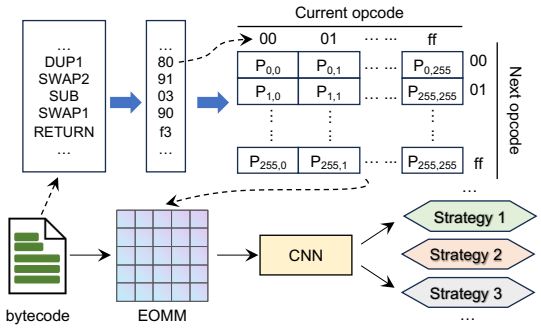

Fig. 5. Analysis process for obfuscated bots.

a multi-classification task. Since no usable training dataset exists, we use currently available tools (cf. Table I) to obfuscate all 172 unobfuscated bots (§V-B-❶). To generate diverse training samples, we employed not only single obfuscation methods from individual tools but also synergistic obfuscation, combining different methods from various tools. This process resulted in 4,216 obfuscated samples, which we labeled with the S_{logic} types. Afterward, the trained classifier is embedded into APOLLO to identify S_{logic} of obfuscated bots.

❷ **S2: Gas optimization.** Gas optimization aims to replace the original sequence with an anti-sequence that maintains the same functionality but requires less gas. Unlike the straightforward task of performing gas optimization in the forward direction, determining whether a contract is gas optimized in reverse is challenging. This is because anti-sequences, such as those in the compiler's built-in optimizations, also appear in the opcode streams of normal contracts, and no features can distinguish them. Therefore, APOLLO focuses on 1) the relationship between the profit and gas cost of MEV bots, and 2) function jump table optimizations that can be explicitly identified, to serve as a lower bound for the analysis of **S2**.

• **Profit and gas analysis.** The average profit and gas cost of each MEV bot are statistically analyzed to reflect the relationship between profitability and gas control.

• **Jump table optimisation.** The bytecode entry has an inherent function jump table structure. When making a contract call, pattern matching is attempted for each function selection fragment in the jump table until it successfully matches a function or reaches the end of the table. Each attempt costs 22 gas. There are currently two techniques for jump table optimization: 1) Jump table order control. Arrange the function selection fragments in correlation with the frequency of function calls, so frequently called functions are matched early in the jump table. 2) Jump table structure modification. Replace the original fragment with a less costly function selection fragment. For example, {`PUSH1 num, EQ, PUSH2 locator, JUMPI`} jumps to the corresponding location `locator` when it matches the custom function number `num`, with an attempt cost of 19 gas. Thus, APOLLO first checks the bytecode for the existence of an inherent jump table structure and then analyzes the jump table gas optimization in two steps. 1) For bots with inherent structure, APOLLO counts the frequency of all functions triggered in historical transactions based on traces, then compares the order of these functions in the jump table and calculates the gas cost required to call these functions through the jump table. 2) For bots without an inherent structure, accurately identifying a custom jump table is challenging. Therefore, APOLLO executes historical transactions opcode by opcode and locates {`PUSH1 0x00, CALLDATALOAD`}, an unchanging pattern used to find the target function from the transaction calldata, indicating the start of the function jump judgment. The transaction execution continues until the first jump, where the target function is successfully matched and jumped. We record the opcodes executed during this period and calculate the gas cost of this function jumping process based on EVM opcode pricing [45].

C. Running Stage Strategy Analysis

1) Opportunities discovery: This step analyses how MEV bots identify suitable token exchange paths as well as look for transactions that may bring them opportunities.

❸ **S3: Identify profitable exchange paths.** APOLLO analyzes MEV bot strategies to identify profitable exchange paths by examining their selection of tokens and token pairs. ① For token, APOLLO identifies token transfer events in MEV transactions by parsing logs [8], [11], [14]. To capture MEV bots' preferences for specific tokens, APOLLO collects the type and percentage of tokens involved in each bot's transactions. Additionally, APOLLO records profits from transactions using each token and collects the market capitalization of each token. This data helps analyze the overall trend between asset size and MEV profits. ② For token pair, APOLLO first collects the from and to addresses in token transfer events, excluding EOA addresses and retaining only contract addresses. It then gathers the relevant contract ABIs and identifies the token pairs involved in the transaction by parsing the token transfer events.

❹ **S4: Search for catalyst transactions.** Our goal of finding catalyst transactions $\mathcal{T}$ is to reveal which transactions provide opportunities for MEV bots and are at risk of being exploited. Intuitively, a "large" transaction that can "greatly" alter the asset price is considered a $\mathcal{T}$ because the resulting price difference can easily trigger a profit opportunity. This empirical insight aligns with financial common sense and has been supported by various studies and reports [8]. However, no prior work has examined whether transactions with other characteristics could also serve as $\mathcal{T}$. We aim to delve deeper into this area to provide more insights.

It is a challenging task to detect unseen transactional relationships from huge blockchain historical data, especially when the analysis goal cannot be predefined. To address this challenge, we design a semi-automatic process for developing detection methods, as shown in Fig. 6, which consists of two steps. ① Association classification (AC), which uses association rule learning (ARL) [46] to identify historical transactions with different potential associations to MEV transactions and classify transactions with similar potential associations into the same cluster. ② Cluster analysis and detection algorithm design, which analyses each transaction cluster manually, focusing only on relevant transactions to discover new transactional relationships with minimal manual effort, and designing the corresponding detection algorithms.

• **Associative classification.** It is a classification learning method in data mining [46]. It first uses ARL, an unsupervised machine learning, to generalize the hidden knowledge (rules) among data, and then constructs classification models (classifiers) after pruning useless and redundant rules. ARL aims to mine the associations between different things in a large database. Compared with other data mining methods [46], ARL is more likely to find hidden associations, and the rules learned by ARL are interpretable, which can help us to perform transactional analyses in ② and further reduce manual work. By using all fields of a transaction as attributes

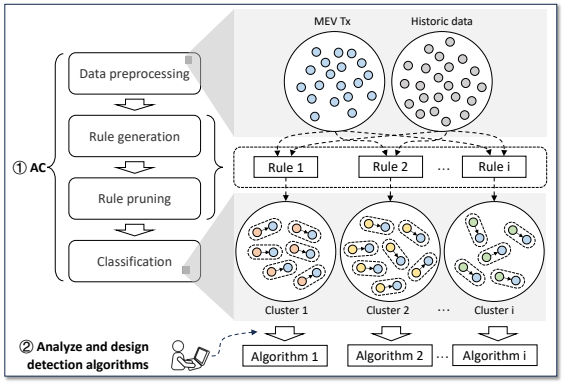

Fig. 6. The process of designing detection methods for **S4**.

to describe a transaction object, ARL can learn a set of rules $\mathcal{R}$ from a historical transaction set H, and an MEV transaction set M, where $\mathcal{R}$ contains multiple rules R. R is in the form of $R : X \Rightarrow Y$, where X is a class of historical transactions in H that satisfy a certain attribute feature, Y is a class of MEV transactions in M that satisfy a certain attribute feature, and there is a potential association between X and Y. Each rule R represents an unknown potential relationship. Since ARL may generate redundant rules, we remove useless rules through a rule pruning process and retain only the high-confidence rules. By applying rule R for classification, historical and MEV transactions with the same potential relationship can be aggregated into a cluster. The exact meaning of this unknown underlying relationship can be known by analyzing the transactions in this cluster. Each rule R in $\mathcal{R}$ generates a cluster. Appendix C details the implemented data processing, rule generation, rule pruning, and classification procedures. Through this process, we obtain 5 clusters.

• **Cluster analysis and detection algorithm design.** We manually analyze transaction clusters to discover new knowledge to explain catalyst transactions. Our design rationale is threefold. First, each cluster contains transaction groups with similar relationships. Analyzing clusters directly is more efficient than blindly exploring all historical transactions. Second, although the classification results of AC have unavoidable false positives, existing studies prove its false positives rate is lower than that of traditional classifiers [46], which is enough to help us effectively reduce the manual effort. Besides, we do not use the results of AC as detection results but only based on analyzing a cluster to discover a new unknown relationship before designing a targeted detection algorithm.

Third, the pruning process narrows the range of attributes of antecedent transactions, making it easier to focus on attributes related to potential relationships and thus discover their significance. For instance, in a cluster of sandwich transactions, the related rules indicate that the $from$ and to addresses and the tokens involved in the antecedent transaction, match those in the subsequent victim transaction, but the transaction $status$ is failed. This suggests that the prior failed transaction may be connected to the MEV transaction. Upon examining the error messages of the failed transactions, we find they failed due to an insufficient return amount. The return amount is a user-defined parameter representing the minimum number of revenue tokens $minreturn$ that the user can accept. If the actual revenue is lower than $minreturn$, the transaction is reverted. Users set slippage protection by specifying $minreturn$, and the level of slippage protection for a transaction is negatively correlated with $\frac{minreturn}{amount}$, where $amount$ is the amount of tokens the user spends. By parsing the calldata of the antecedent and victim transactions, we find that the $\frac{minreturn}{amount}$ of the antecedent transaction is larger than victim transaction. This can be explained psychologically: after a user's profit transaction fails due to slippage protection, they fear missing opportunity and hastily initiate a same transaction except slippage. However, this behavior may provide a signal of slippage failure to attackers focusing on historical failed transactions.

Overall, we find 5 types of catalyst transactions. $\mathcal{T}_1$: Transactions from the same sender following a failed transaction. $\mathcal{T}_2$: Transactions that lack execution chain check. Based on the `CHAINID` instruction, the transaction can check the execution environment blockchain ID (Ethereum mainnet ID is 1, and testnet is other values). Since MEV bots usually judge a pending transaction whether can be exploited by executing it in advance on the testnet. Therefore, transactions involving the execution chain check can be avoided from being inspected, while lack of execution chain check can bring opportunities for bots. $\mathcal{T}_3$: Transactions with excessively high gas prices. Users usually set higher gas prices for high-profit transactions to be confirmed quickly, but such transactions indicate to bots the value of the transaction. $\mathcal{T}_4$: Large transactions. This result confirms previous empirical insights [8]. $\mathcal{T}_5$: Transactions involving altcoins. Based on the characteristics of $\mathcal{T}$ we found, we design targeted detection algorithms in APOLLO. Appendix E provides a detailed description of algorithms.

2) Logic construction: This step introduces the initial funding of the MEV bot and the distribution of profits.

❺ **S5: Initial funds.** To analyze flashloan, for each MEV transaction of a given MEV bot, APOLLO obtains a record of token transfers in its internal transactions. According to the principle of flashloan, if there is an internal transaction in which the lending platform sends tokens to the bot and later the bot sends tokens back to the lending platform, the transaction uses a flashloan. Conversely, it means that the initial funding came from the bot owner.

❻ **S6: Profits allocation.** APOLLO uses the method proposed by Wu et al. to construct the asset flow for each MEV transaction and obtain the address-token state graph [47]. It then considers the address with increased assets as beneficiaries of this MEV transaction.

3) Parameters setting: This step analyses the bot's strategies for controlling risk as well as using parameters.

❼ **S7: Risk tradeoff.** Given an investment, the AMM automatically calculates the revenue based on the token reserves and predefined formulas. Predefined formulas of all AMMs are public, and we can recover any historical token reserves (§A). Thus, for an MEV transaction involving AMMs, we can establish constraints based on the AMM's predefined formula and solve for the theoretical optimal source token

investment OT_0 and the theoretical maximum expected profit $max(\mathbb{E}[P(\mathcal{O})])$ at the time of the transaction. Appendix D provides a detailed explanation and examples of constraint establishment. APOLLO compares OT_0, $max(\mathbb{E}[P(\mathcal{O})])$, to the actual source token investment RT_0, and the actual profit $\mathbb{R}[P(\mathcal{O})]$ of the transactions to reflect the risk trade-off.

❽ **S8: Calldata.** For each bot, APOLLO records the length of $calldata$ for their MEV bot transactions and uses it as a metric to visualise the number of parameters. The relationship between the average length of $calldata$ and profit for each MEV bot is then counted.

❾ **S9: Value.** APOLLO identify **S9** following these steps: (i) Check if the $value$ field in the transaction is nonzero and record it as v. (ii) Retrieve the parameters of functions with depth 1 called by the bots, storing them in list P. Also, store the values of internal functions called within these functions in list V. (iii) Compare v to each item in P and V. If V matches an item in P but does not match any item in V, this indicates that the bot uses $value$ to pass function parameters.

4) Transaction execution: This step analyses the strategies used by bots for transaction sending and submission.

❿ **S10: Builder distribution.** We obtain all the builder addresses from Etherscan [48] and EigenPhi [49], and identify the distribution of each MEV bot's transactions across different builders through address comparison. For transactions that do not involve a builder address, we assume that they are submitted via mempool.

D. Death Stage Strategy Analysis

⓫ **S11: Death.** APOLLO first identifies bots that have stopped being active. If a MEV bot never trades again after a certain point, we consider it abandoned. Then, for each abandoned bot b, APOLLO analyses the profit changes over its lifetime using a two-stage linear regression, a method commonly used in economics to characterize user behavior through profit changes [50]. APOLLO divides b's lifecycle 9:1 into two parts, and then regresses the profit change curves for each of the two parts separately and records their slopes k_1 and k_2. Finally, it records the bots with $k_1 < 0$, $k_2 < 0$ and $k_1 < k_2$, which represents b's overall profit over the lifecycle has been decreasing and decreases more significantly before being abandoned. Additionally, APOLLO groups bots based on their calling addresses to find multiple bots controlled by the same owner. The degree of overlap in the lifecycles of these bots is examined to analyze the patterns of use and abandonment of bots by each owner.

V. EMPIRICAL RESULTS AND EVALUATION

We implement APOLLO in 6,053 lines of Python code and conduct experiments on a server with an Intel Xeon W-1290 CPU (3.2 GHz, 10 cores), and 128 GB memory to answer four research questions: **RQ1:** What strategies are used by MEV bots and what insights can we gain? (§V-B) **RQ2:** Accuracy of APOLLO's business logic strategy analysis? (§V-C) **RQ3:** How do the optimal bot outperform other bots? (§V-D) **RQ4:** What defense and mitigation recommendations are there? (§V-E) **RQ5:** Time overhead of APOLLO? (§V-F)

A. Data collection

MEV bot. We use existing MEV detection methods to identify arbitrage, sandwich, and liquidation transactions [15], [8], [7], which are the most impactful MEVs [8], and categorize the corresponding contracts into three types of bots based on transaction type. In this process, we do not perform additional filtering to ensure data fairness. By collecting data from Jan 2023 to Dec 2024, we collect 1,283 arbitrage, 605 sandwich, and 291 liquidation bots (74 arb-sand, 53 arb-liq, 4 sand-liq, 4 arb-sand-liq bots). A total of 1,295,829 arbitrage, 1,209,139 sandwich, and 11,268 liquidation transactions.

Trace & Log. We use the `debug.traceTransaction()` [51] of the Ethereum archive node to collect transaction traces and the `eth_getLogs()` [52] to collect historical logs.

B. RQ1: Strategy analysis results

❶ **S1: Business logic strategies.** APOLLO identifies 334 obfuscated bots ($\mathcal{B}_{ob}$) and 1,718 unobfuscated bots ($\mathcal{B}_{unob}$). Compared to $\mathcal{B}_{ob}$, all of which have S_{logic}, only 172 of $\mathcal{B}_{unob}$ have S_{logic}. $\mathcal{B}_{ob}$'s average profit was 32.7% higher than $\mathcal{B}_{unob}$'s. This shows that obfuscation can effectively protect the bots' profits. Overall, 506 (24.7%) bots have S_{logic}, indicating that most bots formulate detailed strategies off-chain and pass the execution logic to bots via transactions. Therefore, the bot contract does not have S_{logic} and only acts as an execution agent. Although these bots cannot interact with other contracts during runtime to obtain real-time data, the transaction fees for executing data calculations within the contract are correspondingly reduced. We compare the average profits of ① bots without S_{logic}, ② $\mathcal{B}_{ob}$ with S_{logic}, and ③ $\mathcal{B}_{unob}$ with S_{logic}, and find that ① profitability is lower than ② but higher than ③. Specifically, the average profits of arbitrage/sandwich/liquidation bots in ③ are 57.4%/42.5%/48.2% lower than those of the corresponding types of bots in ①.

For S_{logic}, we find 20 refinement types. Table II shows the strategy distribution and the impact factor (IF). For a strategy s, IF = $\frac{\text{average profit margin of bots with } s}{\text{average profit margin of bots without } s}$, which measures the potential contribution of s to the profitability of the bot. Each strategy except s_8 contributes to the profit margin. Although s_8 cannot increase the profit margin, it can ensure that there will be no negative profit by setting a minimum profit threshold. We observe that for sender checks and recipient determination, many MEV bots use more complex comparison methods such as XOR/shift calculations (s_{11}-s_{13}, s_{16}-s_{19}) compared to regular contracts that typically use hard-coded or calldata direct comparisons [53]. This is because bots typically manage larger assets and prioritize security. Additionally, bots employ unique strategies, such as identifying newly emerging token pairs on AMMs to check for profit opportunities (s_1), pre-calculating reserves after token swaps to predict future gains (s_2), sorting token pairs by liquidity to prioritize checking high-liquidity tokens (s_3), and incorporating built-in thresholds to detect price differences or avoid unprofitable transactions (s_6, s_8). For $\mathcal{B}_{ob}$, flashloan is the most popular strategy, which allows the bot to trade with more appropriate source token inputs without regard to its asset limits. For $\mathcal{B}_{unob}$, hardcoded access

TABLE II
SUMMARY OF BUSINESS LOGIC STRATEGIES

	No	Description	$\mathcal{B}_{unob}$	$\mathcal{B}_{ob}$	IF
Code with business logic strategy	s_1	Find new token pairs on AMMs	16/9.3%	42/12.6%	1.08
	s_2	Expect the reserve of the two tokens after a exchange $x \rightarrow y$	45/26.1%	74/22.2%	1.15
	s_3	Sort the liquidity of token pairs in a fixed list in contracts	14/8.1%	36/10.8%	1.15
	s_4	Interact with a decentralised oracle to get the prices of tokens on different exchanges, e.g., Chainlink	37/21.5%	134/40.1%	1.04
	s_5	Interact with AMM's contract to get token prices, e.g., Uniswap's *getPrice*()	67/40.0%	152/45.5%	1.02
	s_6	Built-in threshold x to determine if the difference between two prices of a token is greater than x	19/11.0%	45/13.5%	1.12
	s_7	Check whether the *calldata* of the transaction is a specific value	24/14.0%	61/18.3%	1.07
	s_8	Built-in threshold x. If the actual profit is less than x% of the expected profit, the transaction is reversed	16/9.3%	71/21.3%	0.76
	s_9	Profit recipient is hardcoded in the MEV bot contract	47/27.3%	22/6.6%	1.04
	s_{10}	Profit recipient is specified by the calldata of the transaction	34/19.8%	106/31.7%	1.13
	s_{11}	Profit recipient is calculated by bitwise XOR of two variables	12/7.0%	54/16.2%	1.11
	s_{12}	Profit recipient is determined by the return value of an external call	13/7.6%	37/11.1%	1.15
	s_{13}	Profit recipient is read from storage	12/7.0%	35/10.5%	1.09
	s_{14}	Set n ratios and transfer proportionally to n profit recipients	19/11.0%	43/12.9%	1.10
	s_{15}	Check whether the transaction sender is a constant value	78/45.3%	20/6.0%	1.02
	s_{16}	Check the transaction sender by XOR it with a constant value	23/13.4%	71/21.3%	1.14
	s_{17}	Check the transaction sender by XOR it with an incoming parameter	13/7.6%	84/25.1%	1.12
	s_{18}	Check the transaction sender by comparing it with the result of an XOR operation	17/9.9%	51/15.3%	1.09
	s_{19}	Check the transaction sender by comparing it with the result of a shift operation	10/5.8%	57/17.1%	1.12
	s_{20}	Interact with a lending platform for flash loans	56/32.6%	205/61.8%	1.19
Code without business logic strategy			1,546	0	-
Total			1,718	334	-

control (s_{15}) is most commonly used, in contrast to $\mathcal{B}_{ob}$'s greater use of complex access control (s_{16}-s_{19}). According to IF, the latter is more conducive to profitability because it is more difficult to identify and prevent illegal account access.

Insight 1: *Obfuscation is currently only used by a small minority of bots and can effectively guarantee the bot's profitability. For bot deployers, using off-chain strategies is more effective than using code-level strategies without protection. Most refinement strategies help improve profitability, while s_8 is more focused on avoiding negative gains and is more suitable for bots that invest in robustness. Flashloan is the most effective method for profitability and is the most popular with $\mathcal{B}_{ob}$. For access control, $\mathcal{B}_{unob}$ prefer hard-coded methods, while $\mathcal{B}_{ob}$ prefer more complex methods based on XOR or shifting.*

❷ **S2: Gas optimization.** Fig. 7 shows the relationship between average gas usage and profit for all bots. It shows that most bots maintain a relatively balanced gas usage and profit, while the more profitable bots exhibit significantly lower gas usage. Conversely, bots with higher gas usage tend to have very low profits. For jump table optimization, APOLLO identifies 802 bots that use the jump table order control. Among them, calls to functions at the top 30% of their function selector accounted for over 78.4% of their total function calls. In addition, modifications to the jump table structure are observed in 103 bots, which use less than 0.1% of their total gas usage for function selection. Among them, we observe a new and most efficient modification as shown in Fig. 8. Instead of using one left opcode sequence for each function, this approach uses one right sequence to select from 16 functions. Additionally, Fig. 7 shows that the number of high-profit transactions is very low. Specifically, transactions yielding profits over $1,000 account for only 0.014% of all transactions and contribute 2.21% of total profits. This indicates that low-profit strategies can compensate through trading volume and generate higher overall returns than rare high-profit strategies.

Insight 2: *Gas optimization can effectively increase profits. 32.1% of bots use jump table order control, which saves an average of 118.4 gas per transaction compared to arranging the jump table in the order of the functions in the contract. For the 6,213,730 transactions involved in these 802 bots, a total of 73,570,563 gas was saved, worth about $10,299 at the Dec 2024 gas price. 4.1% of bots used the jump table structure modification, and among them, we find the optimal way shown in Fig. 8, for a bot with 16 functions, it reduces deployment cost by 74.4% and execution cost by 90.9% at most. High-frequency, low-profit strategies yield higher overall returns than rare high-profit strategies.*

❸ **S3: Identify profitable exchange paths.** Among the token exchange paths of all bots, APOLLO identifies 5,264 tokens involved in arbitrage bots, 28,071 tokens in sandwich bots, and 102 tokens in liquidation bots. Additionally, the results show that transactions involving WETH/USDT/USDC/WBTC/DAI account for 50.3% of all transactions. This indicates that stablecoins pegged to USD and tokens pegged to BTC and ETH are highly favored by bots. The average profit of transactions not involving these tokens is 37.3% lower than those that do.

For token pairs, APOLLO identifies 10,934 token pairs involved in arbitrage, 29,847 pairs in sandwich, and 321 pairs in liquidation bots. According to previous research on path search algorithms [54], a single transaction typically does not

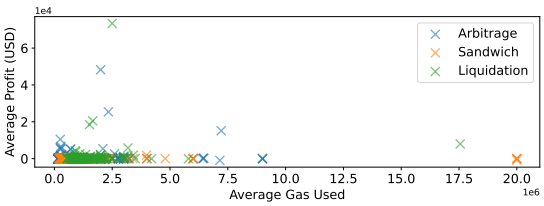

Fig. 7. Relationship between average gas usage and profit

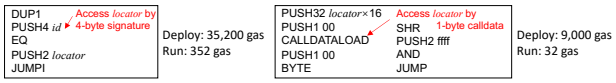

Fig. 8. Optimal jump table modification

TABLE III
CATALYST TRANSACTION STATISTICS

Period	Arbitrage				Sandwich					Liquidation		
	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_4$	$\mathcal{T}_5$	$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_4$	$\mathcal{T}_5$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_4$
P1	13,971	8,923	123,142	11,666	269	14,677	11,042	112,922	8,545	194	305	419
P2	14,693	9,218	140,107	9,814	1,064	14,051	12,286	122,859	6,259	210	339	403
P3	15,256	8,636	134,726	12,357	4,559	14,873	11,743	134,217	7,129	184	298	451
P4	14,739	9,945	151,351	12,702	9,315	15,148	11,369	130,744	6,461	179	327	533

P1: Jan-Jun 2023, P2: Jul-Dec 2023, P3: Jan-Jun 2024, P4: Jul-Dec 2024.

involve more than 5 token pairs, as this would increase the transaction fee cost. However, our results show that 94 (6.4%) of bots have at least one transaction involving more than 10 token pairs. For example, the exchange path of transaction 0x4661 includes 40 token pairs, with small profits gained from each token exchange, and a final profit of $30.

Insight 3: *Token swaps using mainstream tokens are the primary means of MEV bots. Unlike prior research, we find that the current bot can profit by executing long token exchange paths and continuously accumulating small incomes.*

❹ **S4: Search for catalyst transactions.** APOLLO identifies a total of 1,344,620 catalyst $\mathcal{T}$ that offer opportunities. Table III shows each type of $\mathcal{T}$ over time. $\mathcal{T}_4$ is most likely to bring opportunities to bots due to their influence on asset prices. $\mathcal{T}_2$ and $\mathcal{T}_3$ are also easy targets, as transactions lacking `CHAINID`-based execution chain checks cannot avoid being inspected by the bot on the testnet, and transactions with a high gas price tell the bot of their high value. For arbitrage and sandwich bots, $\mathcal{T}_5$ presents a considerable opportunity. This indicates that bots actively monitors emerging altcoins for opportunities. Notably, the opportunities identified by bots in 2024 created by $\mathcal{T}_1$ increased significantly compared to 2023. This indicates that the same transaction initiated by a user immediately after its failed trade is a relatively new $\mathcal{T}$ that bots are paying increasing attention to. For example, after the failure of transaction 0x5e10, the sender re-submits it as 0xf0d9, providing a significant opportunity for a bot to make a profit of $12.9 by spending only $6.8 in costs, achieving a profit rate of 190%. We compare the average profits of bots that can identify $\mathcal{T}$ to those that cannot, and find that the average profit of bots that can identify $\mathcal{T}_1/\mathcal{T}_2/\mathcal{T}_3/\mathcal{T}_4/\mathcal{T}_5$ is 37.7%/20.2%/25.7%/24.4%/28.6% higher.

Insight 4: *Bots that can identify opportunities in catalyst earn 26.3% more than those that cannot. As a new catalyst, $\mathcal{T}_1$ is gaining attention, suggesting that smarter bots will monitor recent history of failed transactions in addition to the previously known monitoring of block and mempool states. Transactions that lack execute chain check and transactions with a gas price set above the market average are at risk. Users should exercise caution when trading in new altcoins.*

❺ **S5: Initial funds.** We detect 902 MEV bots using flashloan, including 722 arbitrage, 95 sandwich, and 83 liquidation bots, involving a total of 1,235,203 transactions. These transactions contribute 56.1% of the total profit, indicating that a significant number of MEV bots currently use flashloan for initial funding and that this strategy is effective in generating profits.

Insight 5: *Using flashloan for initial funding is a widely used strategy and can bring significant benefits to the bot.*

❻ **S6: Profits allocation.** Table IV shows the profit distribution strategies adopted by the three types of bots across five beneficiary types and the total amount of profit received by each type of beneficiary. Overall, 1,143 (55.7%), 685 (33.4%), 1,661 (81.0%), 1,067 (52.0%), and 229 (11.2%) bots distributed profits to bots, transaction senders, builders, validators, and other addresses. The highest number of bots send profits to builders, but by tracking the subsequent profits flow, we find that builders pay 81.6% of their income to validators competing for block creation. However, builder 0x229b only paid 51.5%. It only built 1,175 (0.0005%) blocks but earned 2.34 ETH per block on average. In contrast, the top builder Beaver built 58.4% blocks, paid 94.2% income to validators, and earned only 0.06 ETH per block. After analysis, we find that 0x229b bundle many MEV transactions into a block, enabling it to offer far higher bids than other builders while paying only a small portion of the income. For example, it built four of six consecutive blocks starting from 20622179, where 90.4% of the transactions were MEV transactions. This means that within 48 seconds of these four blocks, only 28 normal user transactions could enter the blockchain.

TABLE IV
BENEFICIARY DISTRIBUTION OF THREE TYPES OF BOTS

Beneficiariy	Arbitrage	Sandwich	Liquidation	Profit
Bot	892	192	59	11.7%
Sender	599	38	48	5.1%
Builder	1,338	215	108	6.4%
Validator	729	206	132	75.4%
Other	183	29	17	1.4%

Overslap is because a bot can allocate profits to multiple beneficiaries.

Insight 6: *Distributing profits to builders is a crucial strategy for bots to distribute direct profit. However, 81.6% of the profits earned by builders were paid to validators. Special builder 0x229b reduce the payment ratio to 51.5% by aggregating many MEV transactions into a single block, but this behavior severely delays normal users' transactions.*

❼ **S7: Risk tradeoff.** We find that 72.4% of the bots, and over 80% of their transactions, do not exchange tokens according to the theoretically optimal input. Despite this, their average profit is 25.9% higher than the bots choosing the theoretically optimal input. For example, Fig. 9 shows the arbitrage bot 0x2490's theoretical investment/profit, and its actual investment/profit, for transactions between Jan and Mar 2023. Theoretical investments and profits are the best deterministic values calculated based on the actual situation at the time. However, as shown, except for a few instances, the bot does not trade strictly according to the theoretical investment. We examine the cases where the theoretical and actual inputs are close and find that these transactions are typically stablecoin-stablecoin swaps or mainstream coin-stablecoin swaps.

Insight 7: *To guarantee a high success rate for transactions, most bots do not strictly adhere to the theoretical optimal investment strategy for token exchanges. However, for transactions involving only stablecoin-stablecoin or mainstream coin-stablecoin exchanges, bots will follow the theoretical optimal investment, as the relative price stability of these assets allows the bots to achieve profits close to the theoretical maximum while maintaining a high success rate.*

❽ **S8: Calldata.** Fig. 10 shows the relationship between average $calldata$ length and profit for the different bots. Bots

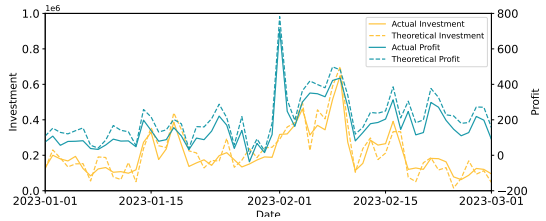

Fig. 9. Theoretical vs actual investment and profit

that rely on off-chain programs to formulate strategies S_{tx} obtain the required parameters directly from the $calldata$ field of the transaction. In contrast, bots with code-level S_{logic} include some parameters in the $calldata$ field, while others are calculated in the contract based on S_{logic}. Consequently, the $calldata$ length for the latter type is naturally smaller than for the former. We combine the results of the analyses in **S1** to observe bots with and without S_{logic} separately. The results show that for bots without S_{logic}, those with higher average $calldata$ lengths tend to have lower profitability, while bots with higher profitability usually exhibit relatively shorter $calldata$ lengths. This suggests that bots with high profitability prioritize optimizing $calldata$ length to reduce gas consumption. For bots with S_{logic}, although they have shorter $calldata$ lengths, they are generally less profitable than bots without S_{logic}, a finding consistent with the analyses in **S1**.

Insight 8: *Reducing calldata length is an effective way to increase profits, but it is not practical for bots with S_{logic}, because the gas increase caused by executing S_{logic} is greater than the gas saved by short calldata.*

❾ **S9: Value.** A total of 5.6% (115) of MEV bots used the strategy of using the $value$ field for calldata compression. Among these bots, 73.4% have higher average profits than the average profit of all bots.

Insight 9: *Using the value field in the transaction to pass parameters is a new way to compress calldata and can improve the profitability of the bot to a certain extent.*

❿ **S10: Builder distribution.** APOLLO identifies 1,084 (84.5%) arbitrage, 520 (85.9%) sandwich, and 235 (80.7%) liquidation bots that chose to submit their transactions via builders. Overall, out of 2,516,236 transactions, 1,224,353 transactions (88.4%) were submitted through 113 different builders. This indicates that MEV bots are more likely to use builders than to submit their transactions directly through the mempool. After analysis, we find that builders with cooperative agreements with validators are more effective in helping bots submit transactions and, consequently, are more prevalent among bots. For example, 22.7% of the bots chose Lido, which operates as both a builder and validator, and 23.4% of transactions were processed through Lido. Additionally, we find that some builders collaborate with bots. For example, builder 0x3B received all transactions in August 2024 from bot 0x64, and this bot only submitted transactions to 0x3B. Through this collaboration, 0x3B earned 1,443 ETH in miner tips.

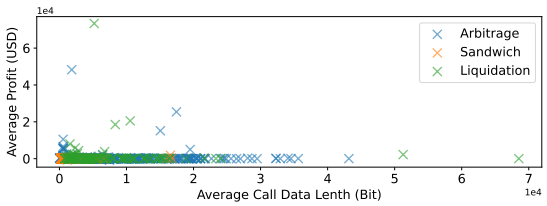

Fig. 10. MEV bot's average calldata length and profit.

TABLE V
AVERAGE PROFIT MARGINS FOR DIFFERENT CONTROL PATTERNS

Pattern	①	②	③	④
Average profit margin	50.5%	512.8%	1108.4%	511.6%

Insight 11: *84.1% of bots use builders because they reduce the risk of transactions being exposed in the mempool. Some builders cooperate with validators or bots to earn more profits.*

⓫ **S11: Death.** APOLLO identifies 452 abandoned MEV bots, of which 62.5% (182/291) arbitrage bots, 59.3% (86/145) sandwich bots, and 93.3% (14/15) liquidation bots have significantly reduced profits at the end of their lifecycle. After analysis, we find that the parameters or contracts of these bots become outdated, resulting in a decline in profitability.

For the patterns of using and abandoning bots by the same owner, APOLLO reveals four types: ① Single bot usage: 1,371 owners use only one bot from beginning to end. ② Simultaneous multiple-bot operation: 560 owners operate multiple bots simultaneously. ③ Sequential bot replacement: 83 owners deprecate a bot after its income decreased and started a new one. ④ Combined strategy: 4 owners employ both simultaneous multiple bot operation and sequential bot replacement strategies. Table V shows the average profit margin of owners in these four patterns. Obviously, ③ is the best strategy, as it ensures continued profitability by abandoning a bot that is no longer profitable and deploying a bot suitable for the new environment. While pattern ④ intuitively has the advantage of ③, the profit margin is unexpectedly lower. Our analysis shows that owners using pattern ④ tend to abandon a bot only after incurring a significant loss, rather than promptly when it starts to decline, leading to a lower average profit margin.

Insight 12: *Outdated parameters or strategies lead to lower profits for MEV bots. At this point, they are usually abandoned. Smarter owners will abandon bots in time and activate new, upgraded bots to ensure their profits.*

C. RQ2: Accuracy of S_{logic} analysis

To evaluate the effectiveness of the APOLLO analysis S_{logic}, we manually construct validation datasets for unobfuscated bots $\mathcal{B}_{unob}$ and obfuscated bots $\mathcal{B}_{ob}$. For $\mathcal{B}_{unob}$, we use state-of-the-art decompilers (Gigahors [55], Slither [56], Rattle [57]) to help decompile all bot bytecode with strategies into readable form to ensure the correctness of strategy analysis. For $\mathcal{B}_{ob}$, it is difficult to inspect all samples because manual bytecode analysis is labor-intensive and cannot be assisted by decompilers. Therefore, we construct a dataset with 800 samples by randomly selecting 20 positive and 20 negative samples for each strategy. For each sample, we first identify the function signatures involved in historical transactions, and then use the EVM Debugger [58] to execute the transactions opcode by opcode to inspect the strategy logic. Four authors collaborated on this analysis to minimize human subjectivity.

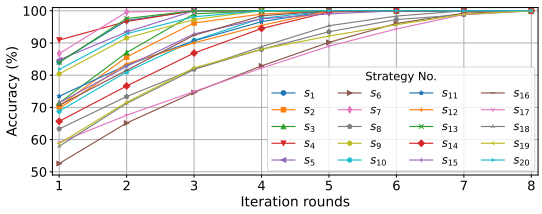

Fig. 11. The accuracy of clustering process

Adjusted Rand Index (ARI). The analysis of $\mathcal{B}_{unob}$ was an iterative process, so we use the ARI to evaluate its performance during the iteration process, which is an indicator of clustering accuracy [59]. As shown in Fig. 11, for simpler strategies such as hard-coded strategies (e.g., s_7 and s_9) and interactive strategies (e.g., s_1 and s_4), complete identification can be within fewer iteration rounds. More complex computational strategies (e.g., s_6 and s_8) and access control strategies (e.g., s_{17}, s_{18}, and s_{19}) require more rounds. Overall, APOLLO can accurately cluster all strategies within 8 iterations.

FP/FN Analysis. For $\mathcal{B}_{ob}$, we analyze APOLLO's false positives (FP: incorrectly reported strategy types) and false negatives (FN: missed strategies) based on the validation dataset. Specifically, for strategy s_2, it has 1 FP and 3 FNs. For s_6, it has 3 FPs and 4 FNs. For s_8, it has 3 FPs and 5 FNs. For s_{13}, it has 3 FNs. For s_{18}, it has 3 FPs and 4 FNs. For s_{19}, it has 2 FPs and 6 FNs. For other strategy types, APOLLO achieves zero FP/FN. After analyzing these FP and FN, we find that all these bots contain incomplete instructions in their bytecode. In EVM, an instruction consists of an opcode and a fixed-length operand (if the opcode requires operands). However, the incomplete instructions in these bots contain only opcodes without operands or have incomplete operands. When APOLLO encounters such instructions, it reads the subsequent data to form a complete instruction, leading to incorrect recognition of the subsequent opcode.

Ablation Study. To evaluate the effectiveness of our obfuscated bot analysis method and its core components (EOMM feature and CNN model), we construct two comparisons: 1) keeping the EOMM features and replacing CNN with LSTM and Transformer. 2) Converting the bot bytecode into a serialized feature vector and using CNN, LSTM, and Transformer for classification. Table VI shows the comparison results of different methods on the validation dataset. Obviously, the EOMM-CNN used by APOLLO outperforms others in all metrics. First, EOMM can characterize the transfer relationship between opcodes that serialized vectors cannot capture, extracting more semantic features from the context. Besides, CNN is better at learning local features from matrix data than LSTM and Transformer, and is more suitable for processing the bytecode transfer relationship features in EOMM.

TABLE VI
PERFORMANCE COMPARISON OF DIFFERENT METHODS

Method	Acc	Precision	Recall	F1
EOMM-CNN	95.4%	97.0%	94.0%	95.8%
EOMM-LSTM	85.8%	90.2%	82.8%	86.3%
EOMM-Transformer	89.1%	91.3%	87.5%	89.4%
Serialized feature-CNN	68.6%	71.8%	67.5%	69.6%
Serialized feature-LSTM	73.5%	76.8%	72.1%	74.4%
Serialized feature-Transformer	74.8%	78.8%	72.9%	75.7%

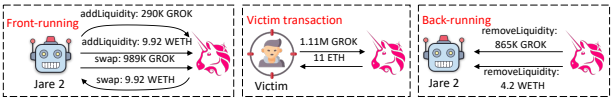

Fig. 12. New sandwich strategy used by the optimal bot

D. RQ3: Case study

We analyze Jared, the highest-earning bot during its operational period. Jared is a sandwich-arbitrage bot that employs obfuscation, preventing existing decompilers from analyzing its bytecode. APOLLO successfully identifies the S_{logic} within it, including s_1, s_2, s_6, s_8, s_{10}, and s_{18}. After analysis, Jared stands out from other bots by employing three unique strategies. First, Jared is willing to pay higher transaction fees. Overall, it allocates 92.9% of its revenue to transaction fees. This indicates that Jared prioritizes transaction success rates over profit margins, aiming to achieve total profit through a high volume of successful transactions. When submitting transactions, it did not concentrate on specific builders, with transactions distributed across 23 different builders. Second, holding a wider variety of tokens. Jared's transactions involved 7,097 tokens, with a maximum of 824 tokens held simultaneously. In contrast, the second-ranked bot held a maximum of 103 tokens simultaneously. Additionally, Jared focuses more on altcoins, using the s_1 to monitor newly emerging tokens in real-time, enabling it to execute transactions involving a wider range of tokens. Third, when altcoin prices rise, Jared not only executes pure sandwich transactions but also purchases the corresponding altcoins in the sandwich's back-running transactions and holds them. By selling them later when prices reach higher points, it maximizes profits.

However, due to MEV competition and rapid development, Jared became outdated and experienced a sharp decline in trading volume and profits starting Aug 7, 2024, dropping to zero by Aug 14. The owner immediately abandons it and activates a new bot Jared2. The new bot inherit Jared's multi-token holding strategy but change the unbiased builder strategy, instead concentrating 64.5% of transactions on Beaver, enabling it to earn more transaction fees. We cannot conclusively state that Jared2 colluded with Beaver, but an interesting finding is that another bot, 0x27, which submitted 98.8% of its transactions to Beaver, shares the same *calldata* with Jared2 (e.g., tx 0x75 and 0xda) and has already transferred 362 ETH to Beaver during this process. This may indicate a potential connection between Jared2 and Beaver. Additionally, Jared2 introduces a more complex new sandwich strategy, as shown in Fig. 12. Unlike the traditional sandwich, which only involves token swaps, the new strategy first adds liquidity for both tokens via front-running transactions, followed by token swaps. After the victim transaction suffers losses, back-running transactions remove liquidity for both tokens.

E. RQ4: Recommendations

Avoid captureable catalyst transactions. As $\mathcal{T}_1$ is increasingly attracting the attention of bots, users should avoid initiating the same token swap transaction immediately after a failed transaction. In addition, for transactions requiring quick

confirmation, users should set a reasonable gas price, which should not be significantly higher than the market average, to avoid attracting bots. Users can use wallets with smart tips, which can prompt the gas setting based on the current network state. For transactions that do not require fast confirmations, users can consider using DeFi with the Time Weighted Average Price (TWAP) algorithm to reduce the possibility of being targeted by bots. By selecting a DeFi with an execution chain check, users can prevent bots from checking their transactions in advance on the testnet by using the `CHAINID` instruction. Besides, users can consider splitting the transaction into multiple small transactions when conducting large transactions. This may increase the transaction fees to a certain extent, but it can effectively balance the risk exposure of a single transaction. Trading in token pairs with deep liquidity can reduce the impact of a single transaction on the market price, lowering the likelihood of sandwich attacks. Moreover, users can use limit orders instead of market orders. Limit orders allow users to set a fixed buy/sell price rather than executing transactions at the current market price, thereby reducing losses due to price fluctuations at the time of execution.

MEV resistance development design. Developers can add optional execution chain check functions to DeFi to ensure the security of transactions for ordinary users who have turned on this function while allowing researchers who have turned off this function to execute transactions on the testnet. In addition, wallet and Dapp developers can integrate MEV protection services (e.g., Blocknative [60]) to send transactions to protected RPC endpoints to protect their users' transactions from being exposed in the mempool. Another suggestion is to consider adding a time lock to the protocol to limit high-frequency trading by bots by storing the timestamp of each user's last transaction and rejecting frequently initiated transactions that violate the time lock period. Moreover, developers can avoid inexperienced users from making hasty and incorrect settings by providing automatic slippage and gas recommendation functions based on the current market status.

F. RQ5: Time overhead

To evaluate the practicality of APOLLO, we measured its time overhead for analyzing MEV bots. Table VII shows the time required to analyze bots with the minimum, maximum, and median number of transactions in each of the three categories of bots, as well as the average number of transactions per bot and the average time overhead for each category. The results indicate a significant correlation between analysis time and the number of MEV transactions. Bots with the fewest transactions across the three categories take only 16.3-17.7s to analyze, while those with the most transactions take 20.6 seconds to 20.3 hours. On average, it takes 2.6 min, 13.3 min, and 17.2 seconds to analyze arbitrage, sandwich, and liquidation bots, respectively. However, this average is skewed by a few bots with an exceptionally high number of transactions, as the distribution of bot transactions is significantly long-tailed (the median being much lower than the mean). For bots with median transaction numbers, APOLLO takes only 16.8s, 17.9s, and 16.4s to analyze arbitrage, sandwich, and liquidation bots. This suggests that for most bots, APOLLO can complete its analysis within several to dozens of seconds. In our experiments, this held true for 83.5% of the bots analyzed.

VI. THREATS TO VALIDITY

Since no ground-truth exists on MEV bot strategies S_{logic}, we manually analyze on-chain bots to summarize the strategy knowledge. We design an iterative clustering learning process to minimize manual work. However, this process is not applicable to $\mathcal{B}_{ob}$ because the model cannot naturally group the obfuscated semantics. Therefore, we analyze $\mathcal{B}_{unob}$ and summarize 20 refined S_{logic}. Although we manually analyze 800 obfuscated samples when constructing the verification dataset and don't find any new strategy types, we cannot guarantee that there are no other types of S_{logic} in unchecked $\mathcal{B}_{ob}$, which may lead to false negatives. In the future, we will put more effort into more in-depth analyses to build more comprehensive datasets to support future research. In addition, the obfuscated train dataset was constructed based on all published tools we can find. Therefore, the trained CNN can only handle the obfuscation techniques used by these tools. If a contract uses a new unpublished obfuscation technique, it may cause APOLLO not to analyze it accurately. Besides, our de-obfuscation is based on EVM opcode-based Markov transition matrix, so APOLLO encounters problems with the few bots we found in §V-C that use incomplete instruction obfuscation due to subsequent opcode identification errors. This motivates our future work on more general smart contract de-obfuscators.

VII. RELATED WORK

MEV analysis. Daian et al. define the MEV concept [6]. Qin et al. are the first to comprehensively quantify MEV regarding arbitrage, sandwich, and liquidation [8]. Zhou et al. evaluate the impact of high-frequency MEV transactions and potential countermeasures [11]. McLaughlin et al. conduct a large-scale study of arbitrage from the perspectives of security, stability, and economy [14]. Additionally, many studies quantitatively analyze the MEV market from various perspectives and obtain meaningful results [12], [13]. Weintraub et al. [15] and Li et al. [5] propose effective MEV detection methods. Zhou et al. propose methods to mitigate MEV [16]. However, these works analyze MEV from a high-level perspective, such as market size and macro impact, and cannot identify the fine-grained strategies of MEV bots, especially the strategy logic embedded in the code and what kind of user transactions would attract bots. In contrast, we analyze the strategies of MEV bots at various stages of their lifecycle to provide more detailed insights into current MEV behavior on the chain.

TABLE VII
TRANSACTION NUMBER AND TIME OVERHEAD

MEV type	Tx numbers/oveahead			Average tx numbers	Average overhead
	Min	Median	Max		
Arbitrage	1/16.5s	13/16.8s	36,945/2.9h	675.2	2.6m
Sandwich	1/17.7s	11/17.9s	243,305/20.3h	1,559.9	13.3m
Liquaidation	1/16.3s	3/16.4s	596/20.6s	16.4	17.2s

Smart contract obfuscation. To protect the business logic of contracts, Zhang et al. designed control flow, data flow, and layout obfuscation methods against the Solidity source code to increase the complexity of contracts [25]. Yan et al. designed four anti-patterns to modify the contract bytecode to disrupt the contract control flow to protect contracts from reverse engineering [27]. Yu et al. designed four obfuscation techniques for contract bytecode, which effectively resisted existing decompilers [26]. The above work successfully brings anti-reversal capability to smart contracts and proves that it is difficult to analyze the obfuscated contracts in the existing work. To comprehensively analyse the strategies of MEV bots, we take the first step towards contractual de-obfuscation.

Smart contract decompilation. There are already many studies dedicated to the reverse engineering of smart contracts. Vandal converts EVM bytecode into IR containing control flow information and combines it with an extensible Datalog specification for contract security analysis [61]. Mythril uses a symbolic execution engine to generate a trace and based on it to generate IR for decompilation [62]. Porosity can decompile EVM bytecode into Solidity source code implemented in C++. EthIR [63] is based on Oyente [32], which converts bytecode into a rule-based representation to reason about the properties of EVM bytecode at a high level. Gigahorse can decompile contract bytecode into a high-level 3-address representation that makes the implicit data and control flow information of the bytecode more explicit [55]. However, these tools can only analyse unobfuscated bytecode and fail with an almost 100% failure rate when applied to obfuscated bytecode [26]. In contrast to them, APOLLO is the first tool that can be used to analyse obfuscated smart contract bytecode.

VIII. CONCLUSION

We conduct the first systematic study of 2,052 MEV bots through the development of APOLLO, a tool for analysing fine-grained strategies across the entire lifecycle of MEV bots. Experimental results show that APOLLO is able to efficiently identify 11 categories of strategies covering a wide range of options for MEV bots. With its aid, we harvested many new insights, especially the application of 20 refined code-level strategies for unobfuscate/obfuscated bots and the analysis of catalyst transactions that may present opportunities for bots that have not yet appeared in existing studies.

ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their constructive comments. This work is partly supported by Hong Kong RGC Projects (PolyU15224121, PolyU15231223), National Natural Science Foundation of China (No. 62332004), and Sichuan Provincial Natural Science Foundation for Distinguished Young Scholars (No. 2023NSFSC1963)

REFERENCES

[1] "Defillama," https://defillama.com/, 2020.
[2] K. Yang, B. Yang, T. Wang, and Y. Zhou, "Zero-cerd: A self-blindable anonymous authentication system based on blockchain," *CJE*, 2023.
[3] Z. He, Z. Li, A. Qiao, J. Li, F. Luo, S. Yang, G. Deng, S. Song, X. Zhang, T. Chen *et al.*, "Maat: Analyzing and optimizing overcharge on blockchain storage," in *FAST*, 2025.
[4] D. Shi, X. Wang, M. Xu, L. Kou, and H. Cheng, "Ress: A reliable and effcient storage scheme for bitcoin blockchain based on raptor code," *CJE*, 2023.
[5] 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 *CCS*, 2023.
[6] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *IEEE SP*, 2020.
[7] C. F. Torres, R. Camino *et al.*, "Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain," in *USENIX Security*, 2021.
[8] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?" in *IEEE SP*, 2022.
[9] 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 *IEEE SP*, 2021.
[10] K. Ko, T. Jeong, J. Woo, and J. W.-K. Hong, "An analysis of crypto gas wars in ethereum," in *APNOMS*, 2022.
[11] L. Zhou, K. Qin, C. F. Torres, D. V. Le, and A. Gervais, "High-frequency trading on decentralized on-chain exchanges," in *IEEE SP*, 2021.
[12] K. Qin, L. Zhou, P. Gamito, P. Jovanovic, and A. Gervais, "An empirical study of defi liquidations: Incentives, risks, and instabilities," in *IMC*, 2021.
[13] Y. Wang, Y. Chen, H. Wu, L. Zhou, S. Deng, and R. Wattenhofer, "Cyclic arbitrage in decentralized exchanges," in *WWW*, 2022.
[14] R. McLaughlin, C. Kruegel, and G. Vigna, "A large scale study of the ethereum arbitrage ecosystem," in *USENIX Security*, 2023.
[15] B. Weintraub, C. F. Torres, C. Nita-Rotaru, and R. State, "A flash (bot) in the pan: measuring maximal extractable value in private pools," in *IMC*, 2022.
[16] L. Zhou, K. Qin, and A. Gervais, "A2mm: Mitigating frontrunning, transaction reordering and consensus instability in decentralized exchanges," *arXiv*, 2021.
[17] G. Morello, M. Eshghie, S. Bobadilla, and M. Monperrus, "Disl: Fueling research with a large dataset of solidity smart contracts," *arXiv*, 2024.
[18] F. Luo, R. Luo, T. Chen, A. Qiao, Z. He, S. Song, Y. Jiang, and S. Li, "Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network," in *ICSE*, 2024.
[19] Z. He, Z. Li, J. Luo, F. Luo, J. Duan, J. Li, S. Song, X. Luo, T. Chen, and X. Zhang, "Auspex: Unveiling inconsistency bugs of transaction fee mechanism in blockchain," in *USENIX Security*, 2025.
[20] F. Luo, H. Lin, Z. Li, X. Luo, R. Luo, Z. He, S. Song, T. Chen, and W. Luo, "Towards automatic discovery of denial of service weaknesses in blockchain resource models," in *CCS*, 2024.
[21] X. Liu, A. Belkhiri, M. Jin, Y. Li, and C. Artho, "Contractviz: Extending eclipse trace compass for smart contract transaction analysis," in *SANER*, 2025.
[22] M. Eshghie and C. Artho, "Oracle-guided vulnerability diversity and exploit synthesis of smart contracts using llms," in *ASE*, 2024.
[23] Z. Li, Z. He, X. Luo, T. Chen, and X. Zhang, "Unveiling financially risky behaviors in ethereum erc20 token contracts," *CJE*, 2025.
[24] "Proposer-builder separation," https://ethereum.org/en/roadmap/pbs/, 2022.
[25] P. Zhang, Q. Yu, Y. Xiao, H. Dong, X. Luo, X. Wang, and M. Zhang, "Bian: Smart contract source code obfuscation," *TSE*, 2023.
[26] Q. Yu, P. Zhang, H. Dong, Y. Xiao, and S. Ji, "Bytecode obfuscation for smart contracts," in *APSEC*, 2022.
[27] W. Yan, J. Gao, Z. Wu, Y. Li, Z. Guan, Q. Li, and Z. Chen, "Eshield: protect smart contracts against reverse engineering," ser. ISSTA, 2020.
[28] "evmcodegen," https://github.com/ethereum/evmcodegen, 2018.
[29] "Mev bot incident analysis," https://www.certik.com/zh-CN/resources/blog/mev-bot-incident-analysis, 2023.
[30] J. Xu and B. Livshits, "The anatomy of a cryptocurrency {Pump-and-Dump} scheme," in *USENIX Security*, 2019.
[31] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.
[32] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *CCS*, 2016.

[33] P. Sirinam, N. Mathews, M. S. Rahman, and M. Wright, "Triplet fingerprinting: More practical and portable website fingerprinting with n-shot learning," in *CCS*, 2019.
[34] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, 2021.
[35] L. Du, Y. Wang, G. Song, Z. Lu, and J. Wang, "Dynamic network embedding: An extended approach for skip-gram based network embedding." in *IJCAI*, 2018.
[36] D. W. Ruck, S. K. Rogers, and M. Kabrisky, "Feature selection using a multilayer perceptron," *Journal of neural network computing*, 1990.
[37] M. Gagolewski, "genieclust: Fast and robust hierarchical clustering," *SoftwareX*, 2021.
[38] R. Xu and D. Wunsch, "Survey of clustering algorithms," *IEEE Transactions on neural networks*, 2005.
[39] R. J. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates," in *PAKDD*, 2013.
[40] A. Likas, N. Vlassis, and J. J. Verbeek, "The global k-means clustering algorithm," *Pattern recognition*, 2003.
[41] Q. Song, H. Huang, X. Jia, Y. Xie, and J. Cao, "Silence false alarms: Identifying anti-reentrancy patterns on ethereum to refine smart contract reentrancy detection," in *NDSS*, 2025.
[42] Z. Ni, C. Wang, J. Tao, and Q. Zhang, "A de-obfuscation system based on Markov models," in *CNSSE*, 2022.
[43] C. Gao, M. Cai, S. Yin, G. Huang, H. Li, W. Yuan, and X. Luo, "Obfuscation-resilient android malware analysis based on complementary features," *TIFS*, 2023.
[44] Z. Ma, M. Jiang, F. Luo, X. Luo, and Y. Zhou, "Surviving in dark forest: Towards evading the attacks from front-running bots in application layer," in *USENIX Security*, 2025.
[45] "Opcodes for the evm," https://ethereum.org/en/developers/docs/evm/opcodes/, 2018.
[46] B. Liu, Y. Ma, and C. K. Wong, "Improving an association rule based classifier," in *PKDD*, 2000.
[47] S. Wu, Z. Yu, D. Wang, Y. Zhou, L. Wu, H. Wang, and X. Yuan, "Defiranger: Detecting defi price manipulation attacks," *TDSC*, 2023.
[48] "Etherscan," https://etherscan.io/, 2020.
[49] "Eigenphi," https://eigenphi.io/, 2022.
[50] D. S. Lee and T. Lemieux, "Regression discontinuity designs in economics," *Journal of economic literature*, 2010.
[51] "debug namespace," https://geth.ethereum.org/docs/interacting-with-geth/rpc/ns-debug, 2013.
[52] "Json-rpc api," https://ethereum.org/en/developers/docs/apis/json-rpc#eth_protocolversion, 2013.
[53] S. Yang, J. Chen, M. Huang, Z. Zheng, and Y. Huang, "Uncover the premeditated attacks: Detecting exploitable reentrancy vulnerabilities by identifying attacker contracts," in *ICSE*, 2024.
[54] K. Kulkarni, T. Diamandis, and T. Chitra, "Routing mev in constant function market makers," in *WINE*, 2023.
[55] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *ICSE*, 2019.
[56] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *WETSEB*, 2019.
[57] "Rattle: Evm binary static analysis framework," https://github.com/crytic/rattle, 2022.
[58] "Evm debugger," https://github.com/0x0abd/Dbgereum, 2021.
[59] M. M. Gösgens, A. Tikhonov, and L. Prokhorenkova, "Systematic analysis of cluster similarity indices: How to validate validation measures," in *ICML*, 2021.
[60] "Mev protection," https://docs.blocknative.com/mev-protection, 2024.
[61] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv*, 2018.
[62] "Mythril: Security analysis tool for evm bytecode," https://github.com/ConsenSys/mythril, 2022.
[63] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *ATVA*, 2018.
[64] Z. He, Z. Liao, F. Luo, D. Liu, T. Chen, and Z. Li, "Tokencat: detect flaw of authentication on erc20 tokens," in *ICC*, 2022.
[65] "moralis," https://moralis.io/, 2020.
[66] J. Ming, D. Xu, L. Wang, and D. Wu, "Loop: Logic-oriented opaque predicate detection in obfuscated binary code," in *CCS*, 2015.
[67] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *VLDB*, 1994.
[68] C. Borgelt, "An implementation of the fp-growth algorithm," in *OSDM*, 2005.
[69] M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li *et al.*, "New algorithms for fast discovery of association rules." in *KDD*, 1997.

APPENDIX A
HISTORICAL DATA RECOVERY

The reason why tokens have a price is that they can be exchanged for stablecoins in the AMM, and this exchange rate determines the price of the token. As mentioned in §II, in the AMM, the exchange rate between tokens is automatically calculated based on the token reserves in the liquidity pool (LP) according to a specific calculation model. Therefore, we first recover the historical data of the token reserves and then recover the historical data of the price based on it.

Token reserve recovery. The token reserve changes with each swap transaction, and each swap operation in the AMM creates a swap log that records the quantity relationship between the two tokens [64]. We first obtain the complete log data from the Archive node, and then filter out the swap logs related to each AMM based on the $topic$ field specific to each AMM. We parse the logs to obtain the token data recorded in them, and for each AMM, we trace its swap logs in chronological order to calculate the token reserves after each historical transaction.

Token price recovery. For a token A, if there is a LP that contains both A and stablecoins (USDT, USDC, etc.), then it can be directly exchanged for stablecoins. If there is no LP consisting of A and stablecoins, then A can be exchanged for other tokens first and then indirectly exchanged for stablecoins. In theory, as long as there is an exchange chain for any token that can ultimately be exchanged for stablecoins, then the price of this token can be derived. However, due to price fluctuations, the longer the exchange chain, the more likely the final result will be inaccurate. In addition, we also consider the following exceptional cases:

- There may be multiple exchange chains for any given token, and multiple prices may be calculated.
- Some newly created LPs do not have enough funds to maintain normal exchanges.
- Since there are malicious tokens and LPs, the prices calculated based on the exchange chains involving them are likely to be inaccurate.

To achieve accurate price calculation, we construct a graph of all tokens and LPs. In the graph, each node represents a token. For any two tokens A and B, if there is a LP that includes both A and B, then A and B are connected by two directed edges $A \to B$ and $B \to A$. We search for exchange chains in the exchange graph according to the following rules for token price calculation:

- Any token can be the starting token of an exchange chain.
- Except for the starting token, the tokens involved in the exchange chain can only include stablecoins and mainstream coins (such as WETH, WBTC, etc.).
- The token at the end of the exchange chain must be a stablecoin.
- The length of the exchange chain is a maximum of 3.

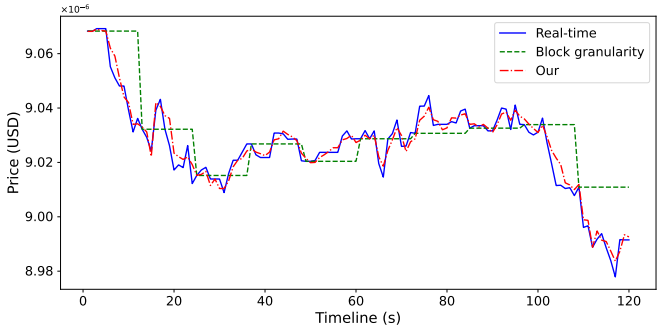

Fig. 13. Price comparison.

To evaluate the accuracy of our designed historical price recovery algorithm, we monitored the real-time price of Pepe (the most traded altcoin on Ethereum) and compared it with the block-granularity price data provided by Moralis [65] and the price calculated based on our method. Fig. 13 shows the results from 14:19:59 to 14:22:59 on July 8, 2024. It is clear that the price curve we recovered is more inaccordance with the real-time price curve than the block-granularity price curve.

APPENDIX B
OBFUSCATION JUDGEMENT METHOD

Based on the existing knowledge of contract obfuscation (§II-D), we design the following method to analyze whether a bot is obfuscated.

- *Bytecode structure.* Contract bytecode has inherent structures. For example, a function jump table contains multiple function select fragments, each consisting of 5 opcodes {`DUP1`, `PUSH4 id`, `EQ`, `PUSH2 locator`, `JUMPI`}, which are used to jump to the `locator` when the function signature `id` is matched. However, obfuscation can alter these inherent structures to interfere with jump analysis, such as using the jump table shown in Fig. 8. Therefore, bytecode with altered inherent structures is considered obfuscated.

- *Control flow.* We identify control flow obfuscated bytecode in 3 steps. 1) Use Gigahors [55] and Slither [56] to construct a control flow graph (CFG) of the bytecode. CFG construction can fail due to timeout or fatal error [55], [26], the former due to exceeding the tool's preset analysis time or loops in the basic block jumps. Therefore, we removed the analysis time limit of the above tools and recorded each newly analyzed jump location, stopping the analysis only if a loop persists at a jump location or if there is a fatal error. Since jump loops or fatal errors are often introduced by obfuscation (nested calls due to code writing errors can also lead to jump loops, but we do not believe that this error exists for MEV bots running continuously in the chain), we consider bots that fail to construct CFGs to be obfuscated. 2) For bots that successfully construct CFGs, we perform symbolic execution. Bots are considered obfuscated if there is unreachable dead code within them. 3) Static symbolic execution struggles to identify false branches introduced by control flow complexity using opaque predicates [66]. Thus for bots that passed the previous step, we execute all their historical transactions opcode by opcode. We identify all conditional comparison opcodes in the transactions, such as `EQ`, `LT`, `GT`, etc., and the associated branch jumping opcodes `JUMP` and `JUMPI`. If there is a branch where the conditions are always the same in all historical executions, we consider it to have a false branch and to be obfuscated.

- *Data flow.* First, construct the data flow graph (DFG). We track all storage accesses (`SLOAD`, `SSTORE`), memory accesses (e.g., `MLOAD`, `MSTORE`, `CALLDATALOAD`, `CODECOPY`), arithmetic operations (e.g., `ADD`, `SUB`, `MUL`), bitwise operations (e.g., `OR`, `XOR`, `SHL`, `NOT`), and hash computations (`KECCAK256`). Connect all access nodes for the same storage slot or memory region to construct the DFG. Then, data flow checks be performed to determine if data flow is obfuscated. For example, if the first operand (slot) of `SSTORE`/`SLOAD` is obtained from `KECCAK256` calculation, it indicates dynamic storage key calculation. If multiple consecutive arithmetic/bitwise operations split and reassemble a variable, it indicates data encoding obfuscation. If data recorded by `LOG` instructions is processed by bitwise operations before being saved, it indicates hidden log/event data obfuscation.

APPENDIX C
ASSOCIATION CLASSIFICATION PROCESS

Suppose $A = \{a_1, \ldots, a_n\}$ is a set of attributes. An object in the dataset can be described by a set of attributes, i.e., the object obj follows the schema $obj = \{attr_1 = a_1, \ldots, attr_m = a_m\}$, $obj \subseteq A$. D is the set of all objects. ARL can learn a rule R form $R : X \Rightarrow Y$ by frequent attribute set mining, where object X is the antecedent, and object Y is the consequent and predetermined goal. This rule indicates an association between X and Y. $sup(R) = P(X \cup Y|D)$, is the support of R, which denotes the probability that both X and Y are included in D. $con(R) = P(Y|X)$, is the confidence of R, which denotes the probability that Y occurs when X occurs. Given a minimum support threshold $minsup$ and a minimum confidence threshold $mincon$, ARL can find the set of frequent attributes that meet the thresholds. To analyze catalyst transactions, we take the following steps.

-I. Data preprocessing. We take all the fields in the transaction as attributes and information about the tokens involved in each transaction, and discretize the continuous values.

-II. Rule generation. At this stage, we test three common ARL algorithms to discover rules, including Apriori [67], FP-Growth [68], and Eclat [69], and finally chose FP-Growth for two reasons. First, it is more efficient in handling large-scale data in our problem. Second, FP-Growth generates a more complete set of rules than the other algorithms so that we can discover more types of unknown relationships [68].

For the two generated rules $R_1 : X_1 \Rightarrow Y$ and $R_2 : X_2 \Rightarrow Y$, if $X_1 \not\subset X_2$ and $X_2 \not\subset X_1$, then R_1 and R_2 represent different types of associations. Otherwise, if $X_1 \subset X_2$, then R_1 and R_2 represent the same association and R_1 is referred to as general rule w.r.t. R_2 and R_2 is referred to as specific rule w.r.t. R_1.

-III. Rule pruning. The number of rules generated by ARL that meet the threshold is huge, and redundant rules are unavoidable. Therefore, we remove useless noise by pruning the

rules. The rationale for our pruning is to consider only high-confidence general rules and prune low-confidence specific rules for the same association. More specifically, suppose there are two rules R_1 and R_2, where R_1 is the general rule w.r.t. R_2. We perform the following checks in turn and when either constraint is satisfied, R_2 will be pruned:

- $con(R_1) > con(R_2)$.
- $con(R_1) = con(R_2)$, but $sup(R_1) > sup(R_2)$.
- $con(R_1) = con(R_2)$ and $sup(R_1) = sup(R_2)$, but $X_1 \subseteq X_2$.

-IV. Classification. We apply the obtained rule set to construct the classifier. We denote a MEV transaction tx_{MEV} and the antecedent transaction tx_{ant} that is potentially associated with it as a transaction group (tx_{ant}, tx_{MEV}). For each class of MEV behavior, the classifier identifies all transaction groups and clusters those with the same potential relationship type into the same cluster. Ultimately, we obtain five clusters.

APPENDIX D
RISK TRADEOFF

Based on the AMM formula and the token reserve, APOLLO obtains OT_0 and $max(\mathbb{E}[P(\mathcal{O})])$ by establishing constraints and solving them. Specifically, take a three-hop arbitrage that occurs between Uniswap V2, Balancer as an example, which has the following three steps: 1) Uniswap V2: $A \rightarrow B$, 2) Balancer: $B \rightarrow C$, 3) Uniswap V2: $C \rightarrow A$, for easy differentiation, we use A_s to denote the source tokens and A_d to denote the destination tokens. Each token LP maintained by Uniswap V2 contains two tokens, which follow the constant product $x \times y = k$, where x and y are the reserves of the two tokens in the LP, and k is a constant. Therefore, for token exchange $x \rightarrow y$, the exchange formula for Uniswap V2 is:

$$[x + (1-p)\Delta x](y - \Delta y) = xy$$

where Δx is the usage of token x, Δy is the income of token y, and p is Uniswap V2's fee, which is 0.3%. Each token LP maintained by Balancer is capable of containing more than two tokens, and each token has a different weight, which follows the free pool formula $\prod_{i=1}^{n} V_i^{W_i} = k$, where V_i denotes the reserve of the i-th token in the LP, W_i denotes the weight of the i-th token, n is the total number of tokens in the LP, and k is a constant. Therefore, for token exchange $x \rightarrow y$, the exchange formula for Balancer is:

$$[x + (1-p)\Delta x]^{W_x}(y - \Delta y)^{W_y} = x^{W_x} y^{W_y}$$

where p is Balancer's fee, which is specific to the LP and ranges from 0.0001% to 10%. Therefore, for the above three-hop arbitrage, APOLLO establishes the following constraints:

$$
\begin{aligned}
&max(\Delta A_d - \Delta A_s)\\
&\text{such that:}\\
&[A_{uni} + (1-p)\Delta A_s](B_{uni} - \Delta B) = A_{uni}B_{uni},\\
&[B_{ban} + (1-p')\Delta B]^{W_B}(C_{ban} - \Delta C)^{W_C} = (B_{ban})^{W_B}(C_{ban})^{W_C},\\
&[C_{uni} + (1-p)\Delta C](A'_{uni} - \Delta A_d) = C_{uni}A'_{uni},\\
&\Delta A_s, \Delta B, \Delta C, \Delta A_d > 0,\\
&\Delta B \leq B_{uni}, \Delta C \leq C_{ban}, \Delta A_d \leq A'_{uni},\\
&A_s \leq A_{current}.(*)
\end{aligned}
$$

where A_{uni}, B_{ban}, and C_{uni} are the initial reserves of the corresponding tokens in the corresponding LPs, respectively, and A'_{uni} is the remaining reserve of token A in the Uniswap LP after performing the token exchange $A \rightarrow B$. $A_{current}$ is the total amount of tokens A available to the MEV bot. In particular, constraint $(*)$ needs to be established on the results of the analysis of the flash loan strategy in **S1** or **S5**. If an MEV bot can use the flash loans as initial funding, it does not need to establish constraint $(*)$, as it does not have to consider the investment quantity constraints. In contrast, bot that can only be funded by the owner has to consider the actual number of assets.

APPENDIX E
DETECTION ALGORITHM

For example, for $\mathcal{T}_1$, Algorithm 1 demonstrates the process of identifying whether the victim transaction in a sandwich transaction is a transaction that was re-initiated after a user's previous transaction failed. The algorithm receives a set of sandwich transactions from an MEV bot and historical transactions from the blockchain. For the victim transaction T_V in each sandwich transaction, the $inputdata$ is decoded to obtain the list of tokens involved in T_V as well as the minimum revenue and number of tokens set by the user, and to identify all failed transactions that execute revert in the 10 blocks preceding T_V (Line 3-4). Then for the failed transaction tx whose $from$ and to are the same as T_V, the $inputdata$ is similarly decoded to obtain the list of tokens, the minimum revenue and the number of tokens (Line 7-8). If tx and T_V involve the same list of tokens and the slippage of tx is lower than that of T_V, it indicates a large degree of correlation between tx and T_V (Line 9-10). Finally, it outputs all identified trades with increased slippage and sandwich transactions.

Algorithm 1: Detect transactions from the same sender following a failed transaction

Input: $sandwich$, a set of sandwich transactions from a bot, and $history$, blockchain history transactions
Output: $\mathbb{A} = [(tx_{slippage}, tx_{san})]$, $tx_{slippage}$ is a transaction that adds slippage hastily, and tx_{san} is a sandwich transaction

1 $\mathbb{A} \leftarrow []$
2 **for** each $tx_{san_i} = (T_{A_1}, T_V, T_{A_2})_i \in sandwich$ **do**
3 $tokenList_{T_V}$, $minReturn_{T_V}$, $amount_{T_V}$ = decodeInput(T_V)
4 $blockRange \leftarrow 10$
5 $\mathbb{T}$ = getRevertedTransaction($T_V, history, blockRange$)
6 **for** each $tx \in \mathbb{T}$ **do**
7 **if** $tx.from = T_V.from$ and $tx.to = T_V.to$ **then**
8 $tokenList_{tx}$, $minReturn_{tx}$, $amount_{tx}$ = decodeInput(tx)
9 **if** $tokenList_{tx} = tokenList_{T_V}$ and $\frac{minReturn_{tx}}{amount_{tx}} < \frac{minReturn_{T_V}}{amount_{T_V}}$ **then**
10 $\mathbb{A}$.append((tx, tx_{san}))
11 return $\mathbb{A}$