BUNDLERLOSS: Financial Attacks on ERC-4337 Bundlers

Bo Lu
Department of Computer Science
George Mason University
Fairfax, VA, United States
blu5@gmu.edu

Xiaokuan Zhang
Department of Computer Science
George Mason University
Fairfax, VA, United States
xiaokuan@gmu.edu

***Abstract*—Account Abstraction (AA) emerged as a solution to overcome the rigid account model limitations of Ethereum by enabling customizable transaction authorization and fee payment mechanisms. ERC-4337, implemented on Ethereum mainnet and EVM-compatible chains (Polygon PoS, Arbitrium one, BNB Smart Chains, etc.), achieves account abstraction through bundlers that aggregate multiple UserOperations from different users into single transactions and submit them to an EntryPoint smart contract. Despite verification safeguards, we identify a critical vulnerability in ERC-4337 that enables malicious actors to inflict financial losses on bundlers.**

In this paper, we introduce BUNDLERLOSS attacks against ERC-4337 bundlers, which exploit the bundling process by having attackers frontrun their own transactions, causing transaction failures that prevent bundlers from receiving gas cost reimbursements. We first outline the workflow of the BUNDLERLOSS attack, study the prerequisites, and investigate EVM-compatible chains to check their susceptibility. Then, we develop mathematical models to quantify the attacker costs and bundler losses. We implement the BUNDLERLOSS attack in local hardhat environment, and perform simulation to demonstrate the correctness of the mathematical models of BUNDLERLOSS. We first simulate our attacks in the testnet, then perform a large-scale simulation on real-world AA transactions, which demonstrates the feasibility and financial impact of these attacks on the ERC-4337 ecosystem. We show that in most cases $1 spent by the attacker can cost over $50 loss by the bundler, and sometimes the attacker can even make profit when launching the attacks.

1. Introduction

Ethereum [1], [2] is a prominent blockchain platform that enables Turing-complete smart contracts for developing decentralized applications (DApps). The Ethereum ecosystem has grown substantially, reaching approximately 230 million unique active wallets and $71 billion in total value locked (TVL) across its DApps as of December 2025 [3]. The platform's functionality extends through layer-2 solutions like Polygon [4], which improve scalability while maintaining Ethereum's security properties. Ethereum-like platforms typically implement two types of accounts: Externally Owned Accounts (EOAs) that initiate transactions but contain no code, and smart contracts that contain code but cannot initiate transactions. However, this rigid design creates several operational constraints. For example, only EOAs can initiate transactions, and users must pay transaction fees (gas) using ETH, Ethereum's native cryptocurrency. These limitations present significant barriers to widespread adoption.

Account Abstraction (AA) emerged as a solution to these constraints by allowing customizable account behavior, including transaction signing, authorization, and fee payment mechanisms. Among various AA proposals [5]–[7], ERC-4337 [7], [8] was implemented and deployed on Ethereum mainnet in March 2023. The ERC-4337 implementation achieves account abstraction without protocol-level changes through four key components: (1) *UserOperation* (`UserOp`): A data structure that replaces traditional transactions, containing operation specifications, signatures, and gas payment details; (2) *Bundler*: An off-chain component that combines multiple `UserOps` into single transactions for submission; (3) *EntryPoint*: A central onchain smart contract that handles `UserOp` verification, execution, and management; (4) *Paymaster*: An optional component that enables third-party gas fee sponsorship. ERC-4337 enables several important capabilities. For example, smart contracts can initiate transactions, users can implement custom authentication methods, and gas fees can be paid by another entity other than the sender, etc. The protocol has demonstrated significant adoption, with over 1.1 billion `UserOps` processed across multiple chains (including Ethereum, Arbitrum, Polygon, and Base) and 58 million active ERC-4337 accounts as of May 2026 [9], [10].

Bundler's economic model. In ERC-4337, bundlers serve as intermediaries by maintaining an offchain alternative mempool specifically for `UserOps`, analogous to how blockchain nodes manage Ethereum's regular mempool. The bundler's primary functions include monitoring this specialized mempool, selecting and verifying valid `UserOps`, and aggregating them into a single transaction for submission to the EntryPoint smart contract on-chain. The bundler's economic model requires them to initially cover the gas costs for transaction execution on Ethereum, which are sub-

sequently reimbursed by either the paymaster or the sender. The bundler's profit is derived from the difference between these reimbursements and their initial gas expenditure. To ensure profitability and mitigate potential losses, bundlers perform local simulation of `UserOps` prior to submission [7], verifying two critical conditions: the successful execution of the `UserOps` and the guarantee of cost reimbursement from either the user or paymaster. Any `UserOps` failing these verification checks are discarded from the mempool.

The BUNDLERLOSS attack. Our research has identified a critical vulnerability in the ERC-4337 protocol that enables malicious actors to inflict financial losses on bundlers, despite the presence of verification safeguards. The vulnerability stems from a fundamental assumption in the bundling process: after successful verification, bundlers presume that transactions containing multiple `UserOps` will be executed successfully on-chain, which is necessary for receiving reimbursement after prepaying the gas fee. However, we demonstrate that a malicious sender can exploit this assumption by frontrunning their own `UserOp`, deliberately causing it to fail, which will revert the entire bundle. As a result, the attack prevents the bundler from receiving their gas cost reimbursement, resulting in direct financial losses. This attack vector represents a novel variation of traditional frontrunning attacks, distinguished by the fact that *the frontrunner and the transaction sender are the same entity (*i.e.*, the attacker)*, rather than the conventional scenario where the transaction sender is the victim. In addition, with the paymaster's involvement, the attack can be *zero-cost*, and sometimes the attacker can even make profits by launching the attack.

BUNDLERLOSS attacks require the same conditions as standard frontrunning attacks, i.e., a public mempool that allows transaction monitoring. To assess the practicality of our attacks, we analyzed popular ERC-4337 compliant blockchains and identified six of them with public mempools that are susceptible to our attacks (§3.3). We also examined the frontrunning activity on EntryPoint contracts v0.8.0 across these blockchains from June to November 2025. Our analysis reveals that over 40% of transactions on Polygon were frontrun during this period, making it an ideal target for evaluating our attacks.

We develop mathematical models based on the official EntryPoint contract to quantify attacker costs and bundler losses. These models capture the gas usage paid by the attacker, the bundler, and the paymaster, providing a theoretical foundation for understanding attack cost-effectiveness. We focus on EntryPoint v0.8.0[1], which was deployed in March 2025 across all ERC-4337 compliant chains, and has over 768,000 transactions in 2025. We implement and evaluate our attacks in a local Hardhat environment running a Polygon node. Our experiments first determine unknown constants in the theoretical models, then examine factors that influence gas usage. When a paymaster is involved, the attacker can achieve negative cost, meaning the attacker earns money while causing bundler losses.

To show the practicality of the attack, we first perform BUNDLERLOSS attacks against our own bundler on the Polygon Amoy Testnet. In addition, we perform large-scale simulation using one month of 1,117 real-world ERC-4337 transactions from Polygon. The results show that in nearly 70% of attacks, the attacker earns a profit while causing bundler losses. In the remaining cases where the attacker does not make a profit, the attacker achieves over 100x cost effectiveness, meaning 1 USD spent by the attacker costs the bundler more than 100 USD. Without paymaster involvement (§6), bundlers lose 50 times more gas costs than attackers spend.

Contributions. We make the following contributions:

- We identify a new attack vector in ERC-4337 called BUNDLERLOSS, where an attacker frontruns their own transaction to cause financial losses to bundlers at zero cost.
- We develop mathematical models that quantify both the attacker's cost and the bundler's loss.
- We evaluate the attack effectiveness through experiments in a local Polygon environment, validating our theoretical models.
- We demonstrate the attack's practicality through testing on the Polygon Amoy testnet, and local simulation of real-world ERC-4337 transactions from Polygon.

Responsible disclosure. We reported our findings to both the Ethereum Foundation and the Polygon Foundation. More details are discussed in §7 and Appendix §A.

2. Background

2.1. Ethereum and Smart Contracts

Ethereum is a decentralized blockchain platform that supports smart contracts through a Turing-complete virtual machine. Smart contracts enable developers to build decentralized applications that execute without centralized control. Ethereum transactions transfer value and invoke contract logic, with execution costs measured in gas units to prevent resource abuse. Since its launch in 2015, Ethereum has become the dominant smart contract platform.

Smart contract is a program deployed on blockchains that automatically executes predefined rules when conditions are met.

Ethereum transaction gas usage consists of multiple components. Every transaction incurs an intrinsic cost of 21,000 gas plus calldata charges (4 gas per zero byte, 16 gas per non-zero byte). The execution cost (opcode gas usage) typically represents the largest portion of total gas consumption.

Opcode Gas Usage reflects the computational cost of executing Ethereum Virtual Machine (EVM) instructions. Each opcode has a specific gas charge based on its resource requirements. Simple operations like arithmetic require few gas units, while memory operations are more expensive

1. While v0.9.0 was deployed in November 2025, it has minimal adoption as of May 1, 2026, with only 14 transactions on Ethereum and 1,138 transactions on Polygon. Therefore, in this paper, we focus on v0.8.0.

Memory expansion cost is the additional gas charged when transactions access memory beyond previously used ranges. This cost follows a quadratic formula, introducing nonlinear components to transaction costs.

Public and Private Transaction Channels By default, a transaction is broadcast to the *public mempool*, where it remains visible to all nodes until it is included in a block, which exposes it to frontrunning. Alternatively, a sender can use a *private channel*, submitting transactions directly to block builders or validators, so that they are not visible before inclusion.

2.2. ERC-4337

ERC-4337 is an Ethereum standard that changes how user accounts work. Instead of being limited to externally owned accounts (EOAs) controlled only by private keys, ERC-4337 introduces a new account model where smart contracts can act as wallets.

UserOperation is a data structure that contains all necessary information for executing a user's intent (similar to a transaction). It specifies the Sender contract address, the target function call, gas limits and fee parameters, extra `preVerificationGas` to pay the bundler, a nonce for replay protection, optional paymaster data for sponsored transactions, and a signature for authorization. Each executed `UserOp` increments the nonce to prevent replay attacks.

EntryPoint contract serves as the central coordinator that receives bundled `UserOps` from bundlers. It first validates each `UserOp` by calling the corresponding Sender contract's validation function. After successful validation of all `UserOps`, the EntryPoint executes each `UserOp` by invoking the Sender contract to perform the requested operations.

Sender contract implements the user's smart contract account that validates `UserOps` and executes the requested operations. Upon successful validation, the Sender contract pays the gas fees for `UserOp` execution and then performs the actions specified in the `UserOp` data. Each Sender contract defines its own validation and execution logic through `validateUserOp` and `execute` functions [11]. The `validateUserOp` function receives a `UserOp` from the EntryPoint Contract and verifies its validity. Users can implement any signature verification method, not limited to traditional ECDSA signatures. The `execute` function receives calldata from the bundler and executes the specified operations accordingly.

Bundler is an off-chain service that collects `UserOps` from users, validates their correctness through simulation, and packages multiple `UserOps` into a single transaction sent to the EntryPoint contract. Bundlers enforce a maximum gas limit per bundled transaction to ensure successful execution within block gas limits.

Paymaster is an optional contract that sponsors gas payments for `UserOps`. When specified in a `UserOp`, the paymaster covers the execution costs instead of the Sender contract, enabling gasless transactions for users. Each paymaster has a

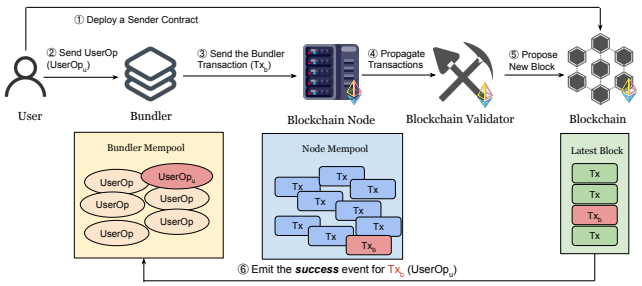


Fig. 1: Normal workflow of ERC-4337.

paymaster policy, which is the developer-defined rules (e.g., per-operation caps, per-user budgets, global spend ceilings, contract allowlists) that set how much gas a paymaster will sponsor. Major providers today such as Pimlico and Alchemy expose the same policy primitives, but keep actual values in private dashboards while users see only terms like "gasless" or "free first mint". Despite this opacity, the aggregate effect is clear: as of May 2026, paymasters have sponsored gas for roughly 935 million UserOperations (85%), paying over $12 million dollars in gas fees for their users [10], [12].

Normal workflow. The normal workflow for ERC-4337 transactions follows the process illustrated in Fig. 1.

- ① The user deploys a Sender contract that implements the required ERC-4337 functions (one example in **??**).
- ② The user submits a $\texttt{UserOp}_u$ to the bundler. The bundler simulates and validates the $\texttt{UserOp}_u$. After validation passes, the $\texttt{UserOp}_u$ is added to the bundler's mempool.
- ③ The bundler packages the `UserOp` (possibly with other `UserOps`) into a transaction (Tx_b), perform additional validations, and submits it to a blockchain node. Tx_b enters the node's mempool.
- ④ Tx_b is propagated through the P2P network.
- ⑤ A blockchain validator includes Tx_b in a newly proposed block following the standard transaction inclusion process.
- ⑥ Upon successful execution, a success event for the `UserOp` is emitted, and the bundler removes all `UserOps` included in Tx_b (including $\texttt{UserOp}_u$) from its mempool.

The money flow is shown in Fig. 2 (Phase I and II).

1. The paymaster deposits money to the EntryPoint contract (**1a**). If there is no paymaster, the user deposits money to the entrypoint contract for the Sender contract (**1b**).
2. After the bundler validates the $\texttt{UserOp}_u$ (e.g.,confirming the Paymaster contract or the sender contract if there is no paymaster specified in the $\texttt{UserOp}_u$ has sufficient funds to pay gas fees), the bundler submits Tx_b to the blockchain. The bundler initially pays the gas fee when Tx_b executes.
3. The EntryPoint contract transfers the gas fee to the bundler to compensate the gas fee paid by the bundler.

2.3. Miner Extractable Value (MEV)

Miner Extractable Value (MEV) [13] refers to the additional revenue that blockchain validators can obtain by manipulating the ordering, inclusion, or exclusion of transactions within a block. Rather than processing transactions in the order they arrive, validators may strategically reorder or insert their own transactions to capture profits from arbitrage, liquidations, or sandwich attacks. A common example is frontrunning [14], where an adversary monitors the mempool for a profitable transaction and submits a higher-fee transaction that executes first, thereby capturing value at the expense of the original user. Prior work has shown that MEV not only distorts incentives for validators but also threatens fairness, market efficiency, and even consensus-layer security in decentralized systems [13], [15].

3. Overview

Threat model. In our attack model, the attacker is a competitor in the AA bundler market, and the attacker's **goal** is to launch BUNDLERLOSS attacks to trigger loss of the target bundler. The attacker does not aim to harm legitimate users. However, legitimate `UserOps` included in the target bundler's transaction are not executed when that transaction reverts. Under the multi-round attack in Appendix §J, their execution may be delayed indefinitely, i.e., they remain in the bundler's mempool and are never executed. The attacker is able to send `UserOps` to the target bundler. The attacker also monitor transactions in the mempool, and runs a MEV bot for launching frontrunning attacks.

Assumptions. We assume that the attacker can send `UserOp` to any bundler, and the attacker also controls MEV bot(s). We assume that the bundler transactions will go through the public mempool. This is evidenced by onchain data reported by BundleBear [16] (a ERC-4337 data analysis platform supported by Ethereum Foundation), showing that a fraction of bundled transactions were frontrun in Table 1 (§3.3). The presence of frontrunning directly indicates that these bundles were visible and exploitable in the public mempool and the MEV bot can observe the transaction sent by the target bundler. Another assumption is that paymasters are willing to cover the cost for the attacker's `UserOps`. We discuss these assumptions in §7. We also relax the paymaster assumption in §6.

Difference with traditional frontrunning attacks. This novel attack vector leverages existing MEV bot infrastructure in an innovative way to achieve its objectives. While traditional MEV operations focus on profit extraction through frontrunning [14], our identified attack methodology is different, as we deliberately position the bundled transaction containing attacker's own `UserOps` as targets for MEV bots. This approach differs fundamentally from conventional MEV in the following aspects: 1) Unlike traditional MEV scenarios where victims are legitimate users, in this attack the "victim transactions" are strategically crafted and deployed by the attacker specifically to trigger frontrunning responses. In other words, the attacker is also the designated "victim"; 2) The goal of the attack is to induce transaction failures to impose financial burdens on bundlers; 3) With the presence of paymaster, a unique ERC-4337 component, the attack can be *zero-cost*.

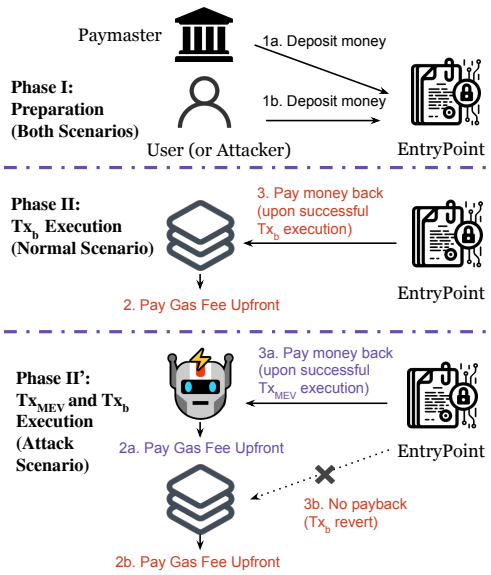


Fig. 2: The money flows: normal vs. BUNDLERLOSS.

3.1. Motivating Example

Consider a typical ERC-4337 transaction flow where a bundler collects multiple `UserOps` from users and packages them into a single transaction for submission to the blockchain. Under normal circumstances, the bundler pays gas fees upfront and recovers costs from the EntryPoint contract upon successful execution. However, an attacker can exploit this process by submitting `UserOps` that appear valid during local validation but are designed to fail onchain (due to frontrunning). The attacker uses MEV bots to frontrun the bundler's transaction, executing only their own `UserOp` in the bundle. When the bundler's transaction subsequently attempts to execute the same `UserOp`, it encounters a nonce conflict and reverts entirely. The bundler loses the gas fee paid for the failed transaction, while the attacker incurs minimal cost (or even zero cost due to paymaster sponsorship).

3.2. Attack Workflow

BUNDLERLOSS attack. We describe the BUNDLERLOSS attack, where the attacker sends one frontrunning transaction. The attack workflow is illustrated in Fig. 3.

- ① The attacker deploys n identical Sender contracts.
- ② The attacker submits n identical `UserOps` (userop_1 to userop_n) to the bundler. After local simulation and validation, these `UserOps` are added to the bundler's mempool.

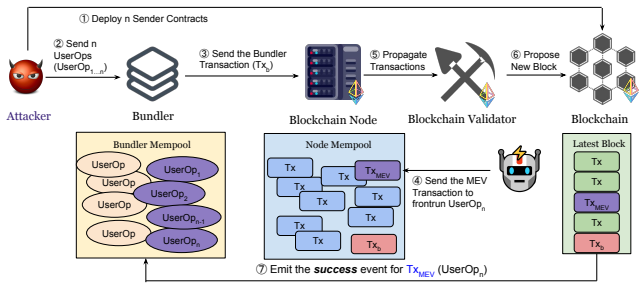


Fig. 3: BUNDLERLOSS attack workflow.

Chain	06/25	07/25	08/25	09/25	10/25	11/25
Avalanche	0/0 (0%)	0/0 (0%)	0/0 (0%)	0/4 (0%)	0/74 (0%)	1/34 (2.94%)
BSC	25/134 (18.66%)	6/142 (4.23%)	0/121 (0%)	1/544 (0.18%)	3/5628 (0.05%)	12/53574 (0.02%)
Celo	0/0 (0%)	0/0 (0%)	0/0 (0%)	0/0 (0%)	0/0 (0%)	0/0 (0%)
Ethereum	7/155 (4.52%)	85/1416 (6.00%)	4/3390 (0.12%)	1/2671 (0.04%)	1/2597 (0.04%)	0/3695 (0%)
Gnosis	5/120 (4.17%)	3/97 (3.09%)	15/65 (23.08%)	11/44 (25%)	0/2 (0%)	1/18 (5.56%)
Polygon	0/4 (0%)	162/361 (44.88%)	121/827 (14.63%)	717/4134 (17.34%)	197/2299 (8.57%)	1841/18293 (10.06%)

TABLE 1: Frontrunned rates for ERC-4337 transactions sent to EntryPoint v0.8.0, from June to November 2025.

- ③ The bundler packages these `UserOps` into a transaction (Tx_b) and submits it to the public Ethereum mempool.
- ④ The attacker's MEV bot monitors the Ethereum mempool. Upon detecting Tx_b, the MEV bot submits a frontrunning transaction (Tx_{MEV}) that contains only the last `UserOp` (userop$_n$) with a higher gas fee.
- ⑤ Both transactions propagate through the P2P network.
- ⑥ A blockchain validator includes both transactions in a new block. Due to its higher gas fee, Tx_{MEV} executes before Tx_b.
- ⑦ Tx_{MEV} executes successfully and emit success event for userop$_n$. However, Tx_b fails when processing the last `UserOp` (userop$_n$), because of duplicated nonce value, as userop$_n$ was already executed of Tx_{MEV}. The failure causes Tx_b to revert completely.

The money flow for the BUNDLERLOSS attack is shown in Fig. 2 (Phase I and II').

1. The paymaster deposits money to the Entrypoint contract (**1a**). If there is no paymaster, the attacker deposits money to the Entrypoint contract for the Sender contract (**1b**).
2. After validating the `UserOps`, the bundler submits Tx_b to the blockchain. The MEV bot observes Tx_b and submits Tx_{MEV} with a higher gas fee.
3. When Tx_{MEV} starts execution, the MEV bot pays the gas fee for Tx_{MEV} (**2a**). When Tx_{MEV} executes successfully, the EntryPoint contract transfers the gas fee to the MEV Bot, which compensates the gas fee paid by the MEV Bot (**3a**).
4. When Tx_b starts execution, and the bundler pays the gas fee for Tx_b (**2b**). However, Tx_b fails at the last `UserOp` because it was already executed in Tx_{MEV}. The bundler loses the gas fee paid for the failed transaction (**3b**).

Design rationale. The attack design follows two key principles. First, all `UserOps` are identical except for the sender field, ensuring uniform gas consumption during execution. Second, only the last `UserOp` is frontrun to minimize the attacker's cost while maximizing the bundler's loss, since the bundler pays for gas usage until the transaction reverts at the final step.

3.3. Practicality of the Attack

We investigate the blockchains that have implemented ERC-4337 and assess whether they are viable targets for our attack. There are 12 EVM-compatible chains that support ERC-4337: Base, Polygon, Worldchain, Arbitrum, Linea, Optimism, Arbitrum Nova, Celo, Avalanche, BSC, Ethereum, and Gnosis. For our attack to be feasible, the Bundler's transaction must pass through a public mempool, allowing our MEV bots to frontrun it. Six of these chains lack a public mempool (Base, Optimism, Arbitrum, Arbitrum Nova, Linea, Worldchain), which makes our attack infeasible. For the remaining chains that are susceptible, we measured the frontrunning rate on each, as reported in Table 1. The frontrunning rate is the percentage of all ERC-4337 transactions that fail due to frontrunning[2]. Celo doesn't have any transaction sent to the EntryPoint v0.8.0 yet. Polygon exhibits a significantly higher frontrunning rate than the others, making it the most suitable chain for our attack.

4. Mathematical Models of BUNDLERLOSS

In this section, we develop mathematical models to quantify the cost-effectiveness of BUNDLERLOSS attacks by analyzing the gas cost of attacker and bundler. Our mathematical models can be applied to all ERC-4337 compliant chains, as long as they follow the official Ethereum reference implementation for the EntryPoint contract. We summarize all notations in Table 2.

4.1. Cost-Effectiveness Analysis

Attacker Gas Fee Cost The attacker controls both the MEV bot and the `UserOp` sender contract. The total attacker cost (G_A) is the same as the gas fee paid by the MEV bot (G_M). When the MEV transaction executes successfully, the EntryPoint contract reimburses the gas fees (paid by the paymaster, G_P) to the MEV bot (Step 3a in Fig. 2), reducing

2. A transaction is failed if EntryPoint reverts. Of the failure codes the EntryPoint reports, only AA25 (nonce already consumed) and AA10 (account already deployed) indicate that another submitter landed the same `UserOp` first; we count these as frontrunning.

Category	Symbol	Represent
Gas Fee Cost (G)	G_M	The MEV bot
	G_U	The sender contract
	G_A	The attacker
	G_B	The bundler
	G_P	The paymaster
Gas Price (Pr)	Pr_M	the MEV transaction
	Pr_B	The bundler transaction
	Pr_P	the paymaster
Total Gas Usage (T)	T_M	The MEV transaction
	T_B	The bundler transaction
	T_P	The paymaster pays for
Opcode Gas Usage (O)	O_M	The MEV transaction
	O_B	The bundler transaction
	O_P	The paymaster pays for
Calldata Gas Usage (D)	D_M	The MEV transaction
	D_B	The bundler transaction
Memory Expansion Gas Usage (E)	E_M	The MEV transaction
	E_B	The bundler transaction
	E_P	The paymaster pays for
Opcode Gas Usage Per `UserOp` (P)	P_M	The MEV transaction
	P_B	The bundler transaction
	P_{PW}	The paymaster pays for
	P_P	$P_{PW} + p$
Number of `UserOps` (N)	N_M	The MEV transaction
	N_B	The bundler transaction
	N_P	The paymaster pays for($N_B = N_P$)
Constants of Gas Usage (c)	c_t	Intrinsic base gas usage of a transaction
	c_m	Constant parts of O_M
	c_b	Constant parts of O_B
	c_{dsf}	Reverted bundler transaction (Eq. 10)
	c_{dmb}	Opcode gas usage difference (P_M-P_B)
	c_p	Constant part of O_P
	c_{dpb}	Opcode gas usage difference (P_P- P_B)
PreverificationGas	p	set by `UserOp` field

TABLE 2: Notations

the MEV cost. Combining these information yields the final attacker cost in Eq. 1.

$$G_A = G_M = T_M \times Pr_M - T_P \times Pr_P \tag{1}$$

Bundler Gas Fee Cost The bundler Gas Fee Cost equals the bundler transaction gas usage (T_B) multiplied by the bundler gas price (Pr_B), as shown in Eq. 2:

$$G_B = T_B \times Pr_B \tag{2}$$

4.2. Insights from the EntryPoint Contract

To analyze the gas usage in detail, we examine the official Ethereum reference EntryPoint implementation. We use EntryPoint v0.8.0 [17] (deployed in March 2025) in our paper, but our method can be applied to future versions. When a bundler receives `UserOps` and packages them into a transaction, it calls the `handleOps` function in the EntryPoint contract (Fig. 4). The total opcode gas usage equals the gas consumed by this function and all functions it invokes. The `handleOps` function performs two main operations: 1) Validation: The `iterateValidationPhase` function (line 11) calls `validateUserOp` in each Sender and `validatePaymasterUserOp` in each paymaster to validate the corresponding `UserOp`. 2) Execution: After all `UserOps` pass validation, the `executeUserOp` function (line 17) calls the `execute` function in each sender contract to perform the requested operations. We utilize this insight to break down the gas usage paid by the MEV bot, bundler and paymaster.

```
1  contract EntryPoint{
2
3      function handleOps(
4          PackedUserOperation[] calldata ops,
5          address payable beneficiary
6      ) external nonReentrant {
7          uint256 opslen = ops.length;
8          UserOpInfo[] memory opInfos = new UserOpInfo
               [](opslen);
9
10         unchecked {
11             _iterateValidationPhase(ops, opInfos,
                   address(0), 0);
12
13             uint256 collected = 0;
14             emit BeforeExecution();
15
16             for (uint256 i = 0; i < opslen; i++) {
17                 collected += _executeUserOp(i, ops[i],
                       opInfos[i]);
18             }
19
20             _compensate(beneficiary, collected);
21         }
22     }
23 }
```

Fig. 4: `handleOps` in the EntryPoint Contract v0.8.0

4.2.1. MEV Bot Total Gas Usage (T_M). The MEV bot's gas usage consists of three items: opcode gas usage O_M, fixed base gas fee of a transaction c_t, and calldata gas D_M:

$$T_M = O_M + c_t + D_M \tag{3}$$

MEV Bot Opcode Gas Usage (O_M) represents the sum of gas costs for the `handleOps` function in the EntryPoint contract and all functions it calls during transaction execution. The `handleOps` function's gas usage can be categorized into two types: constant gas usage that remains fixed regardless of the number of `UserOps`, and variable gas usage that scales with the number of `UserOps`. The constant gas usage includes operations at lines 7, 13, 14, and 20 (Fig. 4), with respective gas costs c_{m11}, c_{m12}, c_{m13}, and c_{m14}. The variable gas usage occurs at lines 8, 11, and 16 as shown in Fig. 4. Each of these operations includes both linear scaling with the number of `UserOps` and memory expansion costs. The gas usage for these operations can be expressed as: $c_{m21} + N_M \times P_{M1} + E_{M1}$, $c_{m22} + N_M \times P_{M2} + E_{M2}$, $c_{m23} + N_M \times P_{M3} + E_{M3}$. Therefore, the total opcode gas usage can be decomposed into three components:

$$c_m = c_{m11} + c_{m12} + c_{m13} + c_{m14} + c_{m21} + c_{m22} + c_{m23} \tag{4}$$

$$P_M = P_{M1} + P_{M2} + P_{M3} \tag{5}$$

$$E_M = E_{M1} + E_{M2} + E_{M3} \tag{6}$$

The total opcode gas usage comprises three parts: a constant component (c_m), a linear component that scales with the number of `UserOps` (P_M), and a memory expansion component (E_M). The per-`UserOp` gas usage(P_M) increases proportionally with the gas consumption of the `validateUserOp`, `execute` functions in the sender contract [11] and `validatePaymasterUserOp` function in the paymaster contract [18]. This will be our first observation as shown in Eq. 7.

Observation 1

$$O_M = c_m + N_M \times P_M + E_M \quad (7)$$

Thus, the total gas usage of MEV transaction is

$$T_M = c_m + N_M \times P_M + E_M + c_t + D_M \quad (8)$$

4.2.2. Bundler Total Gas Usage (T_B). Similarly, for a Bundler transaction, its gas usage also has three parts, opcode gas usage O_B, fixed transaction gas c_t and calldata gas D_B:

$$T_B = O_B + c_t + D_B \quad (9)$$

Bundler Opcode Gas Usage (O_B) consists of the gas cost for the `handleOps` function in the EntryPoint contract and the functions it calls during execution. The bundler transaction execution differs from the MEV transaction because the bundler transaction fails and reverts inside the `iterateValidationPhase` function at line 11 during the last userop execution. As a result, the bundler transaction does not reach the `executeUserop` function at line 17. Only the `validateUserOp` function in the sender contract and `validatePaymasterUserOp` function in the paymaster contract is called.

The bundler transaction still incurs a constant gas usage for line 7, denoted as c_{b11}. The linear part includes only lines 8 and 11. For line 8, the gas usage is $c_{b21} + N_B \times P_{B1} + E_{B1}$. For line 11, since the last userop reverts inside the `iterateValidationPhase` function before completion, there is a constant gas difference c_{dsf} (the details are presented in §E). Moreover, an additional 9300 gas is consumed to access a cold address and its storage, since this is the first time the paymaster contract is called and its storage is accessed by the initial `UserOp`. After we remove this 9300 extra gas, the gas usage for line 11 is $c_{b22} + N_B \times P_{B2} + E_{B2} - c_{dsf}$ when $N_B \geq 2$.

$$c_b = c_{b11} + c_{b21} + c_{b22} - c_{dsf} \quad (10)$$

$$P_B = P_{B1} + P_{B2} \quad (11)$$

$$E_B = E_{B1} + E_{B2} \quad (12)$$

The bundler opcode gas usage has three components: a constant part (c_b), a linear part that scales with the number of userops (P_B), and memory expansion costs (E_B). This leads to our second observation in Eq. 13. The value P_B increases only with the gas usage of the `validateUserOp` function in the sender contract and `validatePaymasterUserOp` function in the paymaster contract, while changes to the `execute` function have no effect.

Observation 2

$$O_B = c_b + N_B \times P_B + E_B, \qquad N_B \geq 2 \quad (13)$$

Thus, the total gas usage of Bundler transaction is

$$T_B = c_b + N_B \times P_B + E_B + c_t + D_B \quad (14)$$

4.2.3. Gas usage to be paid by paymaster (T_P).. T_P is the gas usage that paymaster pays for. It has two parts. One part (p) is set by the each `UserOp` in the `preVerificationGas` field to pay extra gas to Bundler. The extra gas is used to cover calldata cost, intrinsic transaction gas and Entrypoint execution overhead. The total of this part will be $N_P \times p$. The other part is opcode gas usage O_P: By combining these two parts together, T_P will be:

$$T_P = O_P + N_P \times p \quad (15)$$

Paymaster Opcode Gas Usage (O_P) covers three functions (the code snippets are too long to attach): `_validatePrepayment` [19], `innerHandleOp` [20] and `_postExecution` [21]. The gas usage could be divided into two categories: 1) the extra cold address and storage access usages incurred by the first `UserOp`, and 2) the gas usage increases linearly with the number of `UserOps`, as well as memory expansion overhead.

Observation 3

$$O_P = c_p + N_P \times P_{PW} + E_P \quad (16)$$

Let P_P = P_{PW} + p. We have:

$$\begin{aligned} T_P &= c_p + N_P \times P_{PW} + E_P + N_P \times p \\ &= c_p + N_P \times P_P + E_P \end{aligned} \quad (17)$$

4.2.4. Gas Cost. With T_M, T_B and T_P, we can calculate the gas fee for Bundler and attacker:

$$G_B = (c_b + N_B \times P_B + E_B + c_t + D_B) \times Pr_B \quad (18)$$

$$\begin{aligned} G_A = (c_m + N_M \times P_M + E_M + c_t + D_M) \times Pr_M \\ - (c_p + N_P \times P_P + E_P) \times Pr_P \end{aligned} \quad (19)$$

When the gas usage of the `validateUserOp` function increases by Δ, P_B, P_M and P_P increase by Δ.However, when the gas usage of the `execute` function increases by Δ, only P_M and P_P increases by Δ while P_B remains unchanged. Since the sender contract of `UserOp` is controlled by us, we will make `execute` function remain unchanged. Therefore, the difference among the three variables P_B, P_M and P_P will be constant.

Observation 4

$$c_{dmb} = P_M - P_B \quad (20)$$

$$c_{dpb} = P_P - P_B \quad (21)$$

5. Evaluation

In this section, we perform in-depth evaluation to demonstrate the practicality of our attacks, and empirically evaluate the security impacts of our attacks. As Polygon exhibits a significantly higher frontrunning rate than the

others (Table 1), in the evaluation, we concentrate on Polygon. First, we detail our experimental setup for running the evaluations in a local Polygon node using Hardhat (§5.1). Second, to facililate the evaluation, we need to first empirically resolve the constants (*e.g.*, c_m). We use one simple sender contract to measure the constants, and validate them using another sender contract (§5.2). Third, after obtaining the constants, we evaluate the impact factors that impacts the Gas Usage (§5.3). Fourth, we demonstrate the BUNDLERLOSS attacks on the Polygon Amoy testnet against our own bundler (§5.4). Finally, we perform a large-scale simulation by replaying the real-world normal ERC-4337 transactions in Polygon POS from Oct 1, 2025 to Oct 31, 2025, while injecting our attack transactions to demonstrate the practicality of our attacks (§5.5).

5.1. Experimental Setup

We conduct our experiments on a Mac M4 Mini running macOS Sequoia 15.6 with 16GB RAM and 512GB SSD storage. We use Hardhat (version 2.26.1) to run our local Polygon node. We target the EntryPoint contract v0.8.0 [17]. For the bundler, we use the reference bundler implementation (commit: 3706bcc9) [22]. We implement custom MEV bots that continuously monitor the Polygon node mempool for pending transactions, and frontrun bundler transactions by submitting a MEV transaction. The MEV transaction's gas price is set to be 1 unit higher than the bundler transaction, motivated by the observation that 99% of frontrunning transactions on Polygon pay only 1 *wei* more than the frontrunned target transaction.

Sender contracts. We use three sender contracts (X, Y, Z) with different computational complexities to evaluate gas consumption patterns. Contract X (Fig. 5) implements minimal `validateUserOp` and `execute` functions that perform no operations, representing the lowest possible gas consumption baseline. Based on Contract X, Contract Y (Fig. 9 in Appendix §D) adds a simple loop that sums numbers from 0 to 99 in the `validateUserOp` function, increasing gas consumption compared to Contract X. Contract Z (Fig. 10 in Appendix §D) extends this computation by summing numbers from 0 to 199, further increasing gas consumption relative to Contract Y. All `UserOps` use identical field values (nonce, gas limits, and gas fees) except for the sender address and paymaster signature, ensuring consistent calldata gas costs across experiments. Each sender contract uses identical bytecode deployment to guarantee uniform gas consumption per `UserOp` within each contract type.

Paymaster contract. For paymaster contract, we deploy a paymaster contract [23] that is widely used. For example, it was used by 89% of `UserOps` from Oct 1, 2025 to Oct 31, 2025.

Hardhat configuration. We use two Hardhat configurations for different experimental scenarios. For resolving constants and impact factor analysis (§5.2 and §5.3), we initialize Hardhat from an empty blockchain state. For real-world transaction replay (§5.5), we initialized a local fork of the Polygon PoS at the specific block height corresponding to the occurrence of each transaction.

```
contract Sender{
    function validateUserOp(
        UserOperation calldata userOp,
        bytes32 userOpHash,
        uint256 missingAccountFunds
    ) external returns (uint256 validationData) {
       return 0;
    }
    function execute(address dest, uint256 value,
         bytes calldata func) external {
       return;
    }
}
```

Fig. 5: Sender contract X

5.2. Empirically Resolving Constants

To empirically observe the unknown constants c_m, c_b, c_p, c_{dmb} and c_{dpb}, we conduct controlled experiments to empirically determine their values. These constants are computed based on Sender Contract X and verified using Sender Contract Y. Note that c_t = 21000 for all transactions (see §2.1).

5.2.1. Resolving c_m. To determine c_m, we measure O_M and E_M for different values of N_M according to Eq. 7. Table 4 and Table 5 in Appendix §F present the result. Therefore, we have:

$$P_M = \Delta = 52090 \tag{22}$$

$$c_m = O_M - N_M \times P_M - E_M = 18426 \tag{23}$$

5.2.2. Resolving c_b. To determine c_b, we measure O_B and E_B for different values of N_B according to Eq. 13. Table 6 and Table 7 in Appendix §F present the result. Therefore, We have:

$$P_B = \Delta = 45160 \tag{24}$$

$$c_b = O_B - N_B \times P_B - E_B = -20706 \tag{25}$$

5.2.3. Resolving c_p. To determine c_p, we measure O_P and E_P for different values of N_P according to Eq. 17. Table 8 and Table 7 in Appendix §F present the result. Therefore, we have:

$$P_P = \Delta = 93947 \tag{26}$$

$$c_p = T_P - N_P \times P_P - E_P = 9300 \tag{27}$$

5.2.4. Resolving c_{dmb}, c_{dpb}. We get P_M, P_B and P_P from previous two experiments (for contract X). We have:

$$c_{dmb} = P_M - P_B = 52090 - 45160 = 6930 \tag{28}$$

$$c_{dpb} = P_P - P_B = 93947 - 45160 = 48787 \tag{29}$$

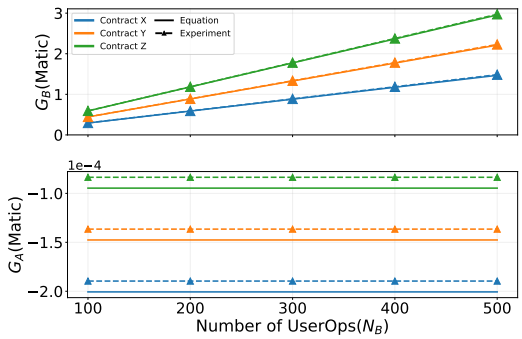


Fig. 6: Gas Fee vs. N_B

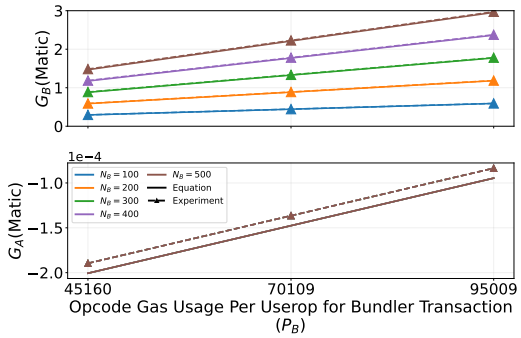


Fig. 7: Gas Fee vs. P_B

5.2.5. Updated Gas Cost. Using the measured constant values from our experiments, we derive the total gas usage for bundler(T_B), MEV transactions(T_M) and the gas usage paymaster pays for (T_P).

Total gas usage formulation. We substitute the measured constants into our gas usage formulas to obtain:

$$T_B = -20706 + N_B \times P_B + E_B + 21000 + D_B \quad (N_B \geq 2) \tag{30}$$

$$T_M = 18426 + N_M \times (P_B + 6930) + E_M + 21000 + D_M \tag{31}$$

$$T_P = 9300 + N_P \times (P_B + 48787) + E_P \tag{32}$$

Gas cost. Since the MEV transaction contains only the last `UserOp` ($N_M = 1$, $N_P = 1$), the updated single-round gas cost becomes:

$$G_B = (-20706 + N_B \times P_B + E_B + 21000 + D_B) \times Pr_B \tag{33}$$

$$G_A = (46356 + P_B + E_M + D_M) \times Pr_M - (58087 + P_B + E_P) \times Pr_P \tag{34}$$

5.2.6. Approximated Gas Fee. We need to approximate the gas fee because the cost of memory expansion grows quadratically with the number of `UserOps`, making the exact mathematical expression very complicated. To make our analysis more tractable, we introduce some approximations.

Pr_B	Pr_M	Pr_P
59,701,132,206	59,701,132,207	57,574,406,342

TABLE 3: Median Gas price values (in *wei*)

Similarly, since calldata costs also increase roughly in proportion to the number of `UserOps`, we apply approximations to the calldata cost as well.

Memory expansion costs. From our experiments, memory expansion costs represent a small fraction of total gas usage. For MEV transactions, E_M accounts for 0.43% to 0.52% of T_M for sender contract X and 0.34% to 0.37% for sender contract Y. For bundler transactions, E_B accounts for 0.42% to 0.43% of T_B for sender contract X and 0.28% to 0.29% for sender contract Y. For paymaster opcode gas usage, E_P accounts for 0.19% to 0.23% of T_P for sender contract X and 0.15% to 0.18% for sender contract Y. Since these costs are negligible, we exclude them from our approximation.

Calldata costs. Since all `UserOps` share identical fields except the sender address and paymaster signature, calldata costs per `UserOp` remain approximately constant. From the data in Table 6 in Appendix §F, D_B grows linearly with N_B and can be expressed as $N_B \times 4000 + 690$. For MEV transactions containing one `UserOp`, D_M equals 4692.

Gas price. We collected all ERC-4337 frontrunning and frontrunned transactions pairs from October 2025 (one month) on Polygon and get the median of (Pr_B, Pr_M, Pr_P) as shown in Table 3. Note that Pr_M is just 1 *wei* greater than Pr_B because 99% of the difference between frontrunnning and frontrunned transactions is 1 *wei*.

Approximate gas usage. By substituting these approximations for memory expansion, calldata costs and gas price, we obtain simplified gas usage formulas:

$$T_B = -20706 + N_B \times P_B + 21000 + N_B \times 4000 + 690 \tag{35}$$

$$T_M = 51048 + P_B \tag{36}$$

$$T_P = 58087 + P_B \tag{37}$$

$$G_B = (-20706 + N_B \times P_B + 21000 + N_B \times 4000 + 690) \times 59701132206 \tag{38}$$

$$G_A = (51048 + P_B) \times 59701132207 - (58087 + P_B) \times 57574406342 \tag{39}$$

Please note that the G_A can be **negative**, meaning the attacker actually earns a profit. This occurs when the paymaster gives more money than the attacker originally spends.

5.3. Impact Factors of G_B and G_A

We use the approximate equations from Eq. 38 and Eq. 39 to analyze the factors that impact G_B and G_A: P_B and N_B. We validate our analysis using three sender contracts (X, Y, Z) with corresponding P_B values of 45160, 70109, and 95009. The G_B and G_A are expressed in *Matic*,

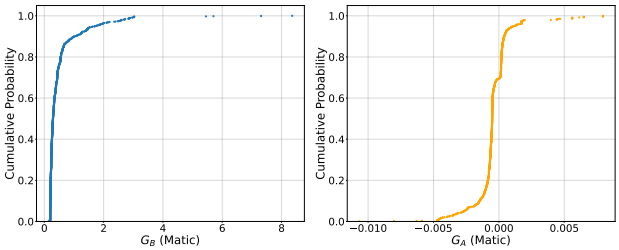


Fig. 8: CDF for G_B and G_A

which is native token of Polygon and 1 *Matic* = 1e18 *wei* = 0.11 USD as of January 1, 2026.

Gas fee vs. N_B. We fix P_B and vary N_B from 100 to 500. Fig. 6 shows the results. According to Eq. 38 and Eq. 39, when P_B is constant, G_A remains constant. The experimental results show small deviations from the theoretical predictions for G_A. This occurs because the gas usage difference between experiments and equations is less than 1% for T_M and T_P. However, when this small difference is multiplied by large gas prices (Pr_M and Pr_P) and then subtracted, the error is amplified, resulting in observable gaps for G_A. In contrast, G_B increases linearly with N_B and involves only multiplication without subtraction, so the small percentage error does not produce significant deviations between theory and experiment. Overall, the theoretical predictions closely match the empirical results. For G_A, the negative values indicate that the attacker generates profit from the attack.

Gas fee vs. P_B. We fix N_B and vary P_B. Fig. 7 presents the results. When N_B is fixed, G_B increases linearly with P_B. For the attacker cost(G_A), when N_B equals 500, it perfect covers other lines when N_B changes from 100 to 400. This is because attacker profit does not change with N_B. Again, the theoretical predictions from Eq. 38 align precisely with the empirical results.

5.4. Polygon Amoy Testnet Simulation

To show our attack could work in real enviroment, we first perform BUNDLERLOSS attack in the Polygon Amoy Testnet [24]. The victim Bundler is run locally using reference bundler implementation (commit: 3706bcc9) [22]. We send 100 attacker `UserOps` to the bundler and use MEV bots to frontrun only the last `UserOp` in the bundle, setting the attacker transaction `maxPriorityFeePerGas` 1 wei higher than the bundle. The bundler and MEV bot transaction hashes are in Appendix §I. Their gas usage is 96,560 and 4,942,421 respectively, only about 0.04% off from our local Hardhat simulation due to calldata differences. The attacker earns about 0.00016 Matic, while the bundler loses about 0.12 Matic, showing that the attacker profits at the bundler's expense.

5.5. Real-World Simulation

To evaluate the attack impact on Polygon, we simulate real-world `UserOps` and inject attacker `UserOps` into bundler transactions. We collect transaction history from the AA EntryPoint v0.8.0 contract on Polygon POS from Oct 1, 2025 to Oct 31, 2025. We observe 2,496 account abstraction transactions in this period. After excluding 1,379 reverted transactions, 1,117 valid transactions remain. We use Contract X as the sender contract for the attacker (P_B = 45,160).

Attack strategy. To replay each transaction, we insert its `UserOps` into the bundler mempool and combine it with attacker `UserOps` such that the resulting bundled transaction includes 100 `UserOps` and uses roughly 5M gas, aligning with the default gas limit of the Ethereum Reference Bundler. We submit bundler transactions at the same block height as the corresponding real-world `UserOp` and use MEV bots to frontrun only the last `UserOp` in the bundled transaction.

Results. 1,117 transactions are simulated to obtain their gas usage. Fig. 8 presents the cumulative distribution function of G_B and G_A. G_B ranges from 0.15 to 8.36 *Matic*. The CDF reaches 96% at $G_B = 2.G_A$ spans from -0.01 to 0.0079 *Matic*. The CDF attains 69% at $G_A = 0$, indicating that in nearly 70% of attacks, the attacker can earn a profit. Even when the attacker incurs a positive cost, the resulting loss for the Bundler can still be substantially larger, which G_B is at least 100 times greater than G_A.

6. BUNDLERLOSS Without Paymaster

In this section, we consider BUNDLERLOSS without paymaster. We update the mathematical models of BUNDLERLOSS, and perform evaluation. Due to page constraints, we present the mathematical model and how we resolve the constants in Appendix §G, and only present our impact factor analysis here.

We use the approximate equations from Eq. 57 and Eq. 58 in Appendix §G to analyze the factors that impact G_B and G_A: P_B and N_B. We validate our analysis using three sender contracts (X, Y, Z) with corresponding P_B values of 36402, 61351, and 86251. The results are shown in Fig. 11a and Fig. 11b in Appendix §H Without paymaster involvement, G_A is always positive, which means attacker could not make profit. The results show that G_B is at least 50 times greater than G_A, demonstrating that attackers can inflict significant bundler losses at minimal cost.

7. Discussion

Limitations. Our model assumes successful frontrunning with a 1-wei gas-fee premium, as observed historically; network conditions and competing searchers may reduce success. We evaluate one bundler implementation because BUNDLERLOSS exploits implementation-independent ERC-4337 properties. Production simulation pipelines, rate limits, and bundle-size policies may affect effectiveness, but we cannot access these configurations or attack production bundlers on mainnet. All attacker `UserOps` are valid and pass standard simulation. We use identical `UserOps` for uniform gas usage; optimizing their diversity requires further

analysis. We evaluate only Polygon, which has the highest frontrunning rate among susceptible chains in Table 1, although ERC-4337 also supports other EVM-compatible chains. For tractability, we omit memory expansion costs, approximate calldata costs linearly, and use median gas prices. The resulting minor errors may be amplified in G_A.

ERC-7562 anti-griefing defenses. ERC-7562 [25] tracks valid `UserOps` received (`opsSeen`) and included on-chain (`opsIncluded`) per entity, throttling or banning entities with low inclusion rates. It cannot prevent BUNDLERLOSS because the attacker's successful execution of $\texttt{UserOp}_n$ counts as inclusion, regardless of who executes it.

Real-world impact. Although we are unable to carry out the attacks themselves for ethical reasons, our evaluation clearly shows their potential impact on the ERC-4337 ecosystem. As Account Abstraction continues to gain traction, now processing millions of transactions each month (across all EntryPoint contract versions), we expect that the consequences of such attacks would become increasingly severe. Building on the single-round attack, we could have a multi-round BUNDLERLOSS attack (Appendix §J).

Paymaster involvement. We assume that the paymaster will be able to sponsor transactions, although this depends heavily on its design and the associated policy. Nonetheless, as demonstrated in the paper, BUNDLERLOSS attacks can still cause substantial losses to the bundler even without any participation from the paymaster, while requiring only minimal cost from the attacker, highlighting the severity of BUNDLERLOSS.

Mitigation. Our attack exploits the visibility of bundler transactions in the public mempool. Bundlers can submit transactions privately through services like Flashbots or operate as block builders themselves. This approach prevents MEV bots from discovering bundler transactions in the public mempool, eliminating the frontrunning opportunity. However, currently, while chains like Polygon has had Flashbots-like private relay support (e.g., MEV bundle relays), it lacks Ethereum's broadly adopted standard, making effective anti–front-running routing less common in practice.

Responsible disclosure. We reported our findings to both the Ethereum Foundation and the Polygon Foundation. Representatives from the Ethereum Foundation clarified that, since most Ethereum transactions are routed through private channels like Flashbots, our attacks would not have a significant effect. Representatives from the Polygon Foundation replied that our attack falls outside their scope.

8. Related Work

In this section, we present the prior work that is most closely related to our work. We also present prior works on smart contract security and Blockchain DoS attacks in Appendix §K.

Research on AA. Current AA research primarily concentrates on three key areas: measurement, security, and privacy. For measurement, Wang *et al.* [26] analyzed AA's operational architecture and proposed an initial security evaluation framework. Lin *et al.* [27] developed classification algorithms and gas measurement techniques for ERC-4337, providing insights into deployment optimization. For security, Fireblocks researchers [28] discovered a critical vulnerability in UniPass's ERC-4337 module enabling account takeover via EntryPoint manipulation. Dimitrov *et al.* [29], [30] examined ZK-based wallet standards and ZK proof integration with AA for enhanced security. For privacy, Chin *et al.* [31] implemented ring signatures for EOA anonymization. Sans and Liu introduced ZK-Team [32] for privacy-preserving shared account management, and Fan *et al.* [33] developed passkey-based authentication frameworks for ERC-4337 wallets.

MEV-related research. Extensive research has quantified MEV extraction in Ethereum transactions [13]–[15], [34]–[37]. Daian *et al.* [13] first proposed and quantified MEV, while Qin *et al.* [15] measured broader BEV including MEV components. Torres *et al.* [14] classified frontrunning into displacement, insertion, and suppression attacks. Wang *et al.* [34] developed frameworks for cyclic arbitrage detection and Capponi *et al.* [35] analyzed MEV's economic impact. Studies on profit distribution found that miners capture most MEV profits [36], [37]. Our work differs by identifying how MEV infrastructure can be weaponized to attack bundlers in ERC-4337 systems.

MEV in ERC-4337. Industry reports describe competing bundlers and MEV bots copying and frontrunning `UserOps` from public bundles to capture reimbursements, causing the original bundles to revert and incur unreimbursed gas costs [38], [39]. These attacks exploit normal users' `UserOps`. By contrast, BUNDLERLOSS submits its own `UserOps` to a target bundler and frontruns them to deliberately cause the bundler's transaction to fail.

9. Conclusion

We presented BUNDLERLOSS, a new attack vector in ERC-4337 where attackers frontrun their own transactions to cause bundler losses. We developed mathematical models to quantify costs and validated them through local Polygon experiments. When a paymaster is involved, attackers can even earn profit while causing bundler losses. We first demonstrated the practicality of our attacks in the testnet, then we simulated attacks using real-world ERC-4337 transactions from Polygon over one month. Results show that 70% of attacks generate attacker profit. In the remaining cases where the attacker does not make a profit, the attacker achieves over 100x cost effectiveness, meaning 1 USD spent by the attacker costs the bundler more than 100 USD. Without paymaster involvement, the attacker can also achieve over 50x cost effectiveness. These findings demonstrate the practical threat of BUNDLERLOSS attacks and the need for enhanced security measures in ERC-4337 to protect bundler infrastructure.

References

[1] V. Buterin, "A next-generation smart contract and decentralized application platform," *white paper*, 2014.

[2] Ethereum, "Ethereum.org," https://ethereum.org/en/, 2026.

[3] DeFillama, "Defillama - ethereum," https://defillama.com/chain/ethereum, 2026.

[4] Polygon Labs, "Polygon," https://polygon.technology, 2026.

[5] Vitalik Buterin et al., "EIP-2938: Account Abstraction," https://eips.ethereum.org/EIPS/eip-2938, 2020.

[6] Sam Wilson et al., "EIP-3074: AUTH and AUTHCALL opcodes," https://eips.ethereum.org/EIPS/eip-3074, 2020.

[7] ERC-4337 Team, "ERC-4337," https://docs.erc4337.io, 2026.

[8] V. Buterin, Y. Weiss, D. Tirosh, S. Nacson, A. Forshtat, K. Gazso, and T. Hess, "ERC-4337: Account abstraction using alt mempool," Ethereum Improvement Proposals, Tech. Rep., 2021.

[9] E. Foundation, "Account abstraction," https://ethereum.org/roadmap/account-abstraction/, 2026.

[10] BundleBear, "ERC-4337 Metrics," https://www.bundlebear.com/erc4337-overview/all.

[11] Ethereum, "Ethereum standard sender contract," https://github.com/eth-infinitism/account-abstraction/blob/develop/contracts/core/BaseAccount.sol, 2025.

[12] BundleBear, "ERC-4337 paymaster Metrics," https://www.bundlebear.com/erc4337-paymasters/all.

[13] 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 *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 910–927.

[14] C. F. Torres, R. Camino, and R. State, "Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain," *arXiv preprint arXiv:2102.03347*, 2021.

[15] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?" *arXiv preprint arXiv:2101.05511*, 2021.

[16] BundleBear, "Bundlebear," https://www.bundlebear.com/erc4337-bundlers/all.

[17] Ethereum, "Ethereum entrypoint contract v0.8.0," https://etherscan.io/address/0x4337084d9e255ff0702461cf8895ce9e3b5ff108#code, 2025.

[18] ——, "Ethereum standard paymaster contract," https://github.com/eth-infinitism/account-abstraction/blob/develop/contracts/core/BasePaymaster.sol, 2025.

[19] ——, "Entrypoint validateprepayment function," https://github.com/eth-infinitism/account-abstraction/releases/tag/v0.8.0, 2025.

[20] ——, "Entrypoint innerhandleop function," https://github.com/eth-infinitism/account-abstraction/releases/tag/v0.8.0, 2025.

[21] ——, "Entrypoint postexecution function," https://github.com/eth-infinitism/account-abstraction/releases/tag/v0.8.0, 2025.

[22] ——, "Ethereum bunlder reference implementation," https://github.com/eth-infinitism/bundler, 2026.

[23] Polygon, "Polygon paymaster contract," https://polygonscan.com/address/0x888888888888ec68a58ab8094cc1ad20ba3d2402#code, 2025.

[24] ——, "Introducing the amoy testnet for polygon pos," https://polygon.technology/blog/introducing-the-amoy-testnet-for-polygon-pos, 2024.

[25] Y. Weiss, D. Tirosh, A. Forshtat, and S. Nacson, "ERC-7562: Account abstraction validation scope rules," Ethereum Improvement Proposals, no. 7562, https://eips.ethereum.org/EIPS/eip-7562, 2023, accessed: Oct. 2026.

[26] Q. Wang and S. Chen, "Account abstraction, analysed," in *2023 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2023, pp. 323–331.

[27] Z. Lin, T. Wang, C. Zhao, S. Zhang, Q. Yang, and L. Shi, "A measurement investigation of erc-4337 smart contracts on ethereum blockchain," in *2024 International Conference on Computing, Networking and Communications (ICNC)*. IEEE Computer Society, 2024, pp. 1164–1170.

[28] O. Yomtov, "Fireblocks researchers uncover first account abstraction wallet," https://www.fireblocks.com/blog/fireblocks-researchers-uncover-first-account-abstraction-wallet-vulnerability, 2023.

[29] I. Dimitrov and W. Prinz, "Innovations in cryptocurrency wallet standards: Analyzing erc4337 and zk-based proposals," in *2024 6th International Conference on Blockchain Computing and Applications (BCCA)*. IEEE, 2024, pp. 566–573.

[30] ——, "The evolution of private keys: Advancing ethereum transaction authentication through zero-knowledge proofs," in *Proceedings of the 3rd Blockchain and Cryptocurrency Conference (B2C'24)*, 2024, pp. 6–8.

[31] K. Chin, K. Emura, and K. Omote, "An anonymous yet accountable contract wallet system using account abstraction," *Journal of Information Security and Applications*, vol. 89, p. 103978, 2025.

[32] T. Sans and D. Liu, "Privacy-preserving account-abstraction for teams on evm chains," in *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2024, pp. 476–484.

[33] X. Fan and X. Yang, "Enabling web2-based user authentication for account abstraction," in *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2024, pp. 9–10.

[34] Y. Wang, Y. Chen, H. Wu, L. Zhou, S. Deng, and R. Wattenhofer, "Cyclic arbitrage in decentralized exchanges," *Available at SSRN 3834535*, 2022.

[35] A. Capponi, R. Jia, and Y. Wang, "The evolution of blockchain: from lit to dark," *arXiv preprint arXiv:2202.05779*, 2022.

[36] J. Piet, J. Fairoze, and N. Weaver, "Extracting godl [sic] from the salt mines: Ethereum miners extracting value," *arXiv preprint arXiv:2203.15930*, 2022.

[37] B. Weintraub, C. F. Torres, C. Nita-Rotaru, and R. State, "A flash (bot) in the pan: Measuring maximal extractable value in private pools," *arXiv preprint arXiv:2206.04185*, 2022.

[38] B. Basche and A. Watts, "On ERC-4337, intents, and MEV," FastLane Labs Blog, https://www.fastlane.xyz/blog/on-4337-intents-and-mev, Apr. 2024, accessed: Oct. 2026.

[39] BlockPI Network, "Why must ERC-4337 bundlers work with block builders?" BlockPI Blog, https://blockpi.io/blog/aa-useroperation-failure-incident-why-is-public-mempool-necessary-to-erc-4337-bundlers-d939d6dc8812, Jul. 2023, accessed: Oct. 2026.

[40] Ethereum, "entrypoint-contract-source-code," https://vscode.blockscan.com/137/0x4337084d9e255ff0702461cf8895ce9e3b5ff108, 2025.

[41] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, 2017.

[42] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016.

[43] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018.

[44] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[45] J. Frank, C. Aschermann, and T. Holz, "{ETHBMC}: A bounded model checker for smart contracts," in *29th USENIX Security Symposium*, 2020.

[46] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2018.

[47] W. Zhang, Z. Zhang, Q. Shi, L. Liu, L. Wei, Y. Liu, X. Zhang, and S.-C. Cheung, "Nyx: Detecting exploitable front-running vulnerabilities in smart contracts," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 146–146.

[48] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018.

[49] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[50] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[51] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[52] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021.

[53] A. Groce and G. Grieco, "echidna-parade: a tool for diverse multicore smart contract fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[54] W. E. Bodell III, S. Meisami, and Y. Duan, "Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains," in *32nd USENIX Security Symposium*, 2023.

[55] M. Salehi, J. Clark, and M. Mannan, "Not so immutable: Upgradeability of smart contracts on ethereum," *arXiv preprint arXiv:2206.00716*, 2022.

[56] N. Ruaro, F. Gritti, R. McLaughlin, I. Grishchenko, C. Kruegel, and G. Vigna, "Not your type! detecting storage collision vulnerabilities in ethereum smart contracts."

[57] M. Zhang, P. Shukla, W. Zhang, Z. Zhang, P. Agrawal, Z. Lin, X. Zhang, and X. Zhang, "An empirical study of proxy contracts at the ethereum ecosystem scale," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 2996–3008.

[58] Y. Pan, W. Han, Y. Duan, M. Zhang, and R. Model, "Collisionrepair: First-aid and automated patching for storage collision vulnerabilities in smart contracts."

[59] S. Yuan, Z. Zhang, J. Lu, D. Sanan, R. Chang, and Y. Zhao, "A complete formal semantics of ebpf instruction set architecture for solana," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA1, pp. 1–27, 2025.

[60] X. Zheng, Z. Wan, D. Lo, D. Xie, and X. Yang, "Why does my transaction fail? a first look at failed transactions on the solana blockchain," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1489–1512, 2025.

[61] S. Smolka, J.-R. Giesen, P. Winkler, O. Draissi, L. Davi, G. Karame, and K. Pohl, "Fuzz on the beach: Fuzzing solana smart contracts," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1197–1211.

[62] Z. Sun, Z. Li, X. Peng, X. Luo, M. Jiang, H. Zhou, and Y. Zhang, "Doubleup roll: Double-spending in arbitrum by rolling it back," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2577–2590.

[63] Z. Li, X. Peng, Z. He, X. Luo, and T. Chen, "famulet: Finding finalization failure bugs in polygon zkrollup," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 971–985.

[64] Z. Sun, X. Luo, and Y. Zhang, "Panda: Security analysis of algorand smart contracts," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1811–1828.

[65] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017.

[66] S. Yang, F. Zhang, K. Huang, X. Chen, Y. Yang, and F. Zhu, "Sok: Mev countermeasures: Theory and practice," *arXiv preprint arXiv:2212.05111*, 2022.

[67] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "Sok: Decentralized finance (defi) incidents," *arXiv preprint arXiv:2208.13035*, 2022.

[68] S. M. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. J. Knottenbelt, "Sok: Decentralized finance (defi)," *arXiv preprint arXiv:2101.08778*, 2021.

[69] L. Heimbach and R. Wattenhofer, "Sok: Preventing transaction reordering manipulations in decentralized finance," in *4th ACM Conference on Advances in Financial Technologies (AFT)*, 2022.

[70] K. Li, Y. Wang, and Y. Tang, "Deter: Denial of ethereum txpool services," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1645–1667.

[71] Y. Wang, Y. Tang, K. Li, W. Ding, and Z. Yang, "Understanding ethereum mempool security under asymmetric {DoS} by symbolized stateful fuzzing," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4747–4764.

[72] Z. Li, Z. Sun, Z. He, J. Chu, H. Zhou, X. Luo, T. Chen, and Y. Zhang, "Denial of sequencing attacks in ethereum layer 2 rollups," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 2084–2098.

Appendix A. Ethical Considerations

Due to ethical considerations, we cannot launch our attacks in real-world blockchain mainnets. All the experiments were done in a local environment on our own laptop and testnet (with minimal impact), and we have never performed our attacks in the mainnet. We have already shared our findings the management organizations of the blockchains (e.g., Ethereum Foundation and Polygon Foundation).

Appendix B. Open Science

To promote transparency and reproducibility in our research, the data artifacts of this paper will be made publicly available, including source code of the BUNDLERLOSS attack implementation, and all data we collected. Our artifact can be found at: https://github.com/SECSAT-LAB-GMU/4337_Bundler_Attack

```
contract Sender{

    function validateUserOp(
        UserOperation calldata userOp,
        bytes32 userOpHash,
        uint256 missingAccountFunds
    ) external returns (uint256 validationData) {
       uint256 sum = 0;
       for (uint256 i = 0; i < 100; i++) {
           sum += i;
       }
       return 0;
    }

    function execute(address dest, uint256 value,
         bytes calldata func) external {
       return;
    }
}
```

Fig. 9: sender contract Y

```
contract Sender{

    function validateUserOp(
        UserOperation calldata userOp,
        bytes32 userOpHash,
        uint256 missingAccountFunds
    ) external returns (uint256 validationData) {
       uint256 sum = 0;
       for (uint256 i = 0; i < 200; i++) {
           sum += i;
       }
       return 0;
    }

    function execute(address dest, uint256 value,
         bytes calldata func) external {
       return;
    }
}
```

Fig. 10: sender contract Z

Appendix C. LLM Usage Statement

AI tools (e.g., ChatGPT) have been used to check grammar and polish the sentences.

Appendix D. Sender Contract Y and Z

The two additional sender contracts are shown in Fig. 9 and Fig. 10.

Appendix E. Details of Calculating c_b

We present the details for calculating c_b (Eq. 10). For line 11 in Fig. 4, since the last userop reverts inside the `iterateValidationPhase` function before completion, there is a constant gas difference c_{dsf}. The gas difference arises because execution terminates at a specific point rather than completing the full validation phase. The reversion occurs at line 786 in the `validateprepayment` function [40] due to nonce mismatch. Between the reversion point and the end of `iterateValidationPhase`, there are lines 790-806 in the `validateprepayment` method and line 333 in the `iterateValidationPhase` function [40]. The opcode gas usage of these lines is constant because it does not depend on the number of userops (N_B) or the sender contract implementation. Therefore, the gas usage for line 11 in Fig. 4 is $c_{b22} + N_B \times P_{B2} + E_{B2} - c_{dsf}$.

N_M	O_M	D_M	$T_M(mea)$	$T_M(eq)$	E_M	$O_M - E_M$	$\Delta(=P_M)$	c_m
1	70932	4692	96624	96624	416	70516	-	18426
2	123338	8672	153010	153010	732	122606	52090	18426
3	175746	12676	209422	209422	1050	174696	52090	18426
4	228156	16656	265812	265812	1370	226786	52090	18426
5	280567	20660	322227	322227	1691	278876	52090	18426

TABLE 4: MEV gas usage with Sender contract X. *mea*: measured value; *eq*: expected value from equations.

N_M	O_M	D_M	$T_M(mea)$	$T_M(eq)$	E_M	$O_M - E_M$	$\Delta(=P_M)$	c_m
1	95881	4692	121573	121573	416	95465	-	18426
2	173236	8672	202920	202908	732	172504	77039	18426
3	250593	12676	284269	284269	1050	249543	77039	18426
4	327952	16656	365608	365608	1370	326582	77039	18426
5	405312	20660	446972	446972	1691	403621	77039	18426

TABLE 5: MEV gas usage with Sender contract Y. *mea*: measured value; *eq*: expected value from equations.

Appendix F. Resolving Constants

We list the tables mentioned in §5.2 here, including Table 4, Table 5, Table 6, Table 7, Table 8, Table 9.

Appendix G. Mathematical Model without Paymaster

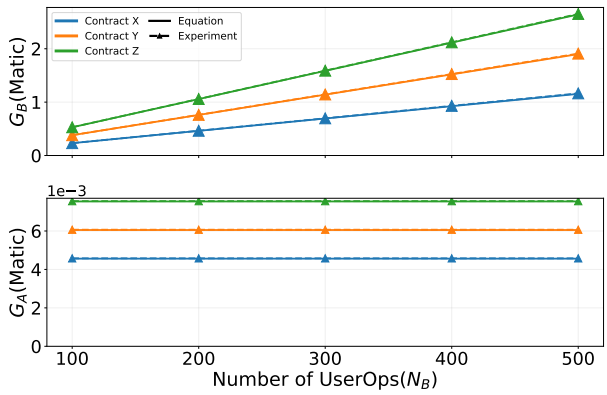
G.1. Cost-effectiveness Analysis

In the following, we update the equations from §4.1.

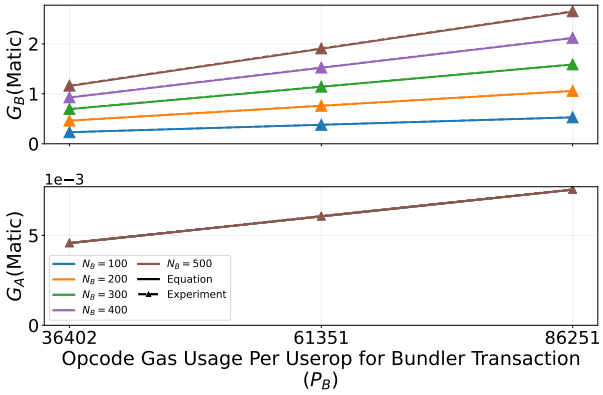
Attacker Gas Cost. Since we do not have paymaster here, the sender contract will pay for the part that is supposed to be paid by paymaster. The attacker controls both the MEV bot and the `UserOp` sender contract. The total attacker cost (G_A) consists of gas fees paid by the MEV bot (G_M) and the sender contract (G_U), as shown in Eq. 40. Note that we do not consider the one-time deployment cost of the sender contract, which we discuss in §7. When the MEV transaction executes successfully, the EntryPoint contract reimburses the gas fees to the MEV bot (Step 3a in Fig. 2), reducing the MEV cost (Eq. 41). Combining these equations yields the final attacker cost in Eq. 42.

$$G_A = G_M + G_U \tag{40}$$

$$G_M = T_M \times Pr_M - G_U \tag{41}$$



(a) Gas fee vs. N_B



(b) Gas Fee vs. P_B

Fig. 11: Gas fee (without paymaster).

N_B	O_B	D_B	$T_B(mea)$	$T_B(eq)$	E_B	$O_B - E_B$	$\Delta(=P_B)$	c_b
1	15360	4692	41052	41052	206	15154	-	-20706
2	70038	8672	99710	99710	424	69614	54460	-20706
3	115418	12676	149094	149094	644	114774	45160	-20706
4	160800	16656	198456	198456	866	159934	45160	-20706
5	206183	20660	247843	247843	1089	205094	45160	-20706

TABLE 6: Bundler gas usage with Sender contract X. *mea*: measured value; *eq*: expected value from equations.

N_B	O_B	D_B	$T_B(mea)$	$T_B(eq)$	E_B	$O_B - E_B$	$\Delta(=P_B)$	c_b
1	40309	4692	66001	66001	206	40103	-	-20706
2	119936	8672	149608	149608	424	119512	79409	-20706
3	190265	12676	223941	223941	644	189621	70109	-20706
4	260596	16656	298,252	298252	866	259730	70109	-20706
5	330928	20660	372588	372588	1089	329839	70109	-20706

TABLE 7: Bundler gas usage with Sender contract Y. *mea*: measured value; *eq*: expected value from equations.

$$G_A = T_M \times Pr_M \tag{42}$$

Bundler Gas Cost. The bundler loss equals the bundler transaction gas usage (T_B) multiplied by the bundler gas price (Pr_B):

$$G_B = T_B \times Pr_B \tag{43}$$

Gas Fee:

$$G_B = T_B \times Pr_B; G_A = T_M \times Pr_M \tag{44}$$

G.1.1. MEV Bot Total Gas Usage (T_M). The MEV bot's gas usage consists of three components: opcode gas usage O_M, fixed base gas fee of a transaction c_t, and calldata gas D_M:

$$\begin{aligned} T_M &= O_M + c_t + D_M \\ &= c_m + N_M \times P_M + E_M + c_t + D_M \end{aligned} \tag{45}$$

G.1.2. Bundler Total Gas Usage (T_B). Similarly, for a Bundler transaction, its gas usage also has three parts, opcode gas usage O_B, fixed transaction gas c_t and calldata gas D_B: O_B follows Eq. 13:

$$T_B = O_B + c_t + D_B = c_b + N_B \times P_B + E_B + c_t + D_B \tag{46}$$

Similarly, the difference between P_M and P_B is a constant.

$$c_{dmb} = P_M - P_B \tag{47}$$

N_P	T_P	E_P	$T_P - E_P$	$\Delta(=P_P)$	c_p
1	103486	239	103247	-	9300
2	197607	413	197194	93947	9300
3	291728	587	291141	93947	9300
4	385851	763	385088	93947	9300
5	479974	939	479035	93947	9300

TABLE 8: Paymaster gas usage with Sender contract X.

N_P	T_P	E_P	$T_P - E_P$	$\Delta(=P_P)$	c_p
1	128435	239	128196	-	9300
2	247505	413	247092	118896	9300
3	366575	587	365988	118896	9300
4	485647	763	484884	118896	9300
5	604719	939	603780	118896	9300

TABLE 9: Paymaster gas usage with Sender contract Y.

G.2. Resolving Unknown Constants

We conduct same experiments (§5.2) to empirically determine c_m, c_b, and c_{dmb}. These three constants are computed based on Sender Contract X and verified using Sender Contract Y.

$$P_M = \Delta = 43313 \tag{48}$$

$$c_m = O_M - N_M \times P_M - E_M = 9126 \tag{49}$$

$$P_B = \Delta = 36402 \tag{50}$$

$$c_b = O_B - N_B \times P_B - E_B = -16120 \tag{51}$$

$$c_{dmb} = P_M - P_B = 43313 - 36402 = 6911 \tag{52}$$

Gas Fee:

$$T_B = -16120 + N_B \times P_B + E_B + 21000 + D_B \quad (53)$$

$$T_M = 9126 + 1 \times (P_B + 6911) + E_M + 21000 + D_M \quad (54)$$

Approximated Gas Fee:

$$T_B = -16120 + N_B \times P_B + 21000 + N_B \times 2272 + 700 \quad (55)$$

$$T_M = 9126 + 1 \times (P_B + 6911) + 21000 + 2972 \quad (56)$$

$$G_B = (-16120 + N_B \times P_B + 21000 + N_B \times 2272 + 700) \times 59701132206 \quad (57)$$

$$G_A = (9126 + 1 \times (P_B + 6911) + 21000 + 2972) \times 59701132207 \quad (58)$$

Other Costs for Attacker: Our mathematical models exclude initial deployment costs required before launching the attack. The attacker must deploy one contract per `UserOp` and deposit funds to the EntryPoint contract for each sender contract. For Sender contract X, deployment costs 225,240 gas and depositing funds requires 45,654 gas. The total deployment cost per `UserOp` is 270,894 gas. Attacker contracts can be reused across multiple attack rounds, amortizing deployment costs over time. Deployment costs can be reduced using EIP-7702 to link a single sender contract to multiple Externally Owned Accounts (EOAs). This approach requires only 25,000 gas per EOA, reducing deployment costs by approximately 90%.

Appendix H. Gas fee vs N_B and P_B

We list figures mentioned in §6, including Fig. 11a and Fig. 11b

Appendix I. Amoy Testnet Attack Transaction Hashes

Bundler transaction hash:
`0x1c89e66606c00980e07ddfff4d61693a3067973b88ce420c160f980fc108c162`
MEV bot transaction hash:
`0xeac7f0f25107115d9330838fc4c8aa3b258f6bf22202f9e218628424adfefa41`

Appendix J. Multi-Round Attack

We can combine multiple single-round attacks in a loop to launch multi-round attacks. After the first round attack completes, the remaining UserOps remain in the bundler mempool because only the last UserOp was executed by the MEV transaction. The attacker can then submit a new UserOp with an incremented nonce (previous nonce + 1) that targets the same sender contract as the last UserOp from the previous round. The bundler will bundle these remaining UserOps together with the new UserOp and submit another transaction to the blockchain. The attacker's MEV bot repeats the same frontrunning strategy, submitting a transaction containing only the last UserOp with a higher gas fee. This process can continue indefinitely, with each round following the same pattern: the bundler transaction fails, the MEV transaction succeeds, and the bundler loses gas fee. Our experiments show that the total costs of both the bundler and the attacker grow almost linearly with the number of rounds. This is because the per-round costs remain nearly constant, apart from minor differences in calldata and storage costs. Therefore, after k rounds, the bundler's loss and the attacker's cost are approximately $k \times G_B$ and $k \times G_A$, respectively.

Appendix K. Additional Related Work

Research on smart contract security. Smart contract security research focuses on identifying vulnerabilities before deployment through systematic analysis of contract code and attack patterns on Ethereum. Static analysis approaches [41]–[47] employ symbolic execution and formal verification to detect vulnerabilities and bugs in Ethereum smart contracts. Fuzzing-based approaches [48]–[53] dynamically test smart contracts to discover previously unknown bugs. Recent work has examined security issues specific to upgradable smart contracts [54]–[58]. Security analysis has also expanded to other blockchain platforms including Solana [59]–[61], Arbitrum [62], Polygon [63], and Algorand [64]. Given the significant impact of smart contract vulnerabilities and DeFi attacks, researchers have produced comprehensive systematization studies [65]–[69] to organize knowledge in this domain. Our work differs from these efforts by specifically examining security vulnerabilities in ERC-4337 systems.

Blockchain DoS attacks. Prior work has examined DoS attacks against Ethereum nodes or Layer 2 sequencers. Li *et al.* [70] show that carefully crafted transactions can severely disrupt Ethereum mempool at low cost. Wang *et al.* [71] demonstrate that Ethereum mempools are vulnerable to asymmetric DoS attacks and introduce stateful fuzzing to systematically uncover such vulnerabilities. Li *et al.* [72] propose a new denial of sequencing attack that disrupts the liveness of layer 2 rollups at zero cost. Unlike these works that aim to paralyze the blockchain node or sequencer, BUNDLERLOSS does not constitute a DoS attack. Instead, BUNDLERLOSS selectively causes targeted bundler transactions to fail, resulting in financial losses for specific bundlers.