USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Auspex: Unveiling Inconsistency Bugs of Transaction Fee Mechanism in Blockchain

Zheyuan He, *University of Electronic Science and Technology of China;* Zihao Li, *The Hong Kong Polytechnic University;* Jiahao Luo, *University of Electronic Science and Technology of China;* Feng Luo, *The Hong Kong Polytechnic University;* Junhan Duan, *Carnegie Mellon University;* Jingwei Li and Shuwei Song, *University of Electronic Science and Technology of China;* Xiapu Luo, *The Hong Kong Polytechnic University;* Ting Chen and Xiaosong Zhang, *University of Electronic Science and Technology of China*

https://www.usenix.org/conference/usenixsecurity25/presentation/he-zheyuan

This paper is included in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

Auspex: Unveiling Inconsistency Bugs of Transaction Fee Mechanism in Blockchain

Zheyuan He[†], *Zihao Li*[‡*], *Jiahao Luo*[†], *Feng Luo*[‡], *Junhan Duan*[§], *Jingwei Li*[†*], *Shuwei Song*[†], *Xiapu Luo* [‡], *Ting Chen*[†], *Xiaosong Zhang*[†*]
[†]*University of Electronic Science and Technology of China*
[‡]*The Hong Kong Polytechnic University* [§]*Carnegie Mellon University*

Abstract

The transaction fee mechanism (TFM) in blockchain prevents resource abuse by charging users based on resource usage, but inconsistencies between charged fees and actual resource consumption, termed as *TFM inconsistency bugs*, introduce significant security and financial risks.

In this paper, we present Auspex, the first tool that automatically detects TFM inconsistency bugs in Ethereum ecosystem by leveraging fuzzing technology. To efficiently trigger and identify TFM inconsistency bugs, Auspex introduces three novel technologies: (i) *a chain-based test case generation strategy* that enables Auspex to efficiently generate the test cases; (ii) *a charging-guided fuzzing approach* that guides Auspex to explore more code logic; and (iii) *fee consistency property and resource consistency property*, two general bug oracles for automatically detecting bugs. We evaluate Auspex on Ethereum and demonstrate its effectiveness by discovering 13 previously unknown TFM inconsistency bugs, and achieving 3.5 times more code branches than state-of-the-art tools. We further explore the financial and security impact of the bugs. On one hand, these bugs have caused losses exceeding millions of dollars for users on both Ethereum and BSC. On the other hand, the denial-of-service (DoS) attack exploiting these bugs can prolong transaction wait time by 4.5 times during the attack period.

1 Introduction

The blockchain [59] is a decentralized ledger, that records the account information (e.g., fund) of the users [48]. The users can send transactions to transfer their funds or invoke a predefined program (i.e., smart contract) [72]. To mitigate spam transactions and resource abuse, the blockchain adopts a transaction fee mechanism (TFM) [41], which charges transaction fees to the users according to the cost of the transaction's resources, involving network, storage, and computation [61].

*Corresponding authors

TFM inconsistency bugs. TFM inconsistency bug is a type of vulnerability characterized by a discrepancy between the fees charged to users and the actual computational resources utilized. These TFM inconsistency bugs can lead to overcharging or undercharging, thereby undermining the security and stability of the system. Specifically, overcharging results in users incurring higher fees than necessary, leading to financial losses [42]. For instance, when users access the same object (account or contract variable) multiple times in a transaction, the corresponding object is cached in memory. However, Ethereum will overcharge those memory accesses as disk accesses, and cause funding loss for users, which has been fixed in EIP-2929 [15]. Besides, undercharging can be exploited by adversaries to launch DoS attacks, thereby compromising the service of blockchain [61]. In 2016, adversaries exploited the undercharged IO-intensive operations to attack Ethereum, and caused severe congestion [14]. Furthermore, TFM inconsistency bugs always have widespread impacts, as TFM of Ethereum has been reused by over 600 blockchains [7], e.g., Binance smart contract (BSC) [5], Polygon [13], Avalanche [4], and Optimism [11].

Limitation. Several works [33, 37, 61, 74] attempt to study TFM inconsistency bugs, but they cannot find the TFM inconsistency bugs comprehensively. These studies often manually design benchmarks for each opcode (i.e., the low-level virtual machine instruction). By analyzing the benchmarks' results, researchers determine whether the fee for the current opcode matches its execution time. For example, Brokenmetre [61] designs the benchmark to find the opcode with heavy execution time. However, their approaches cannot detect TFM inconsistency bugs comprehensively because of three limitations: (i) *Inefficiency in test case generation.* The benchmarks rely on manual case construction, which is inefficient in constructing the test case. (ii) *Inadequacy in considering all the scenarios.* The manual benchmark tests cannot cover all possible scenarios, leading to incomplete detection (e.g., most existing works only focus on opcodes, ignoring fund transfers and precompiled contracts [10, 72]). (iii) *Lack of automation analysis.* Manually analyzing the results of benchmarks is

difficult and subjective, hindering the scalability and reproducibility of the detection.

Our Work. In this work, we propose Auspex, the first detection framework to automatically uncover TFM inconsistency bugs in blockchain systems. Auspex applies fuzzing technology [66] to handle the limitations of inefficiency, inadequacy, and lack of automation. Specifically, Auspex consists of three key techniques. (i) To efficiently generate test cases, Auspex adopts a *chain-based strategy* that can construct a complete blockchain, including chain configurations, blocks, transactions, and contract operations (§4.1). The chain configurations indicate the parameters that execute the blocks (e.g., a series of cache sizes). Compared with the manually crafted benchmark, the approach enables Auspex to efficiently generate more test cases. (ii) To adequately cover the blockchain implementation, Auspex wields a *charging-guided fuzzing approach* that innovatively takes two new pieces of feedback, besides code coverage, to guide the fuzzing process (§4.2). We consider the relationship between gas charge types and their resource consumption as fuzzing feedback, which can better characterize the dynamic behavior of TFM. Besides, we also take the inconsistency (e.g., gas charges, code coverage, resource consumption) between two blockchain instances with different chain configurations as the fuzzing feedback, which can cover more inconsistent implementation in TFM. (iii) To automatically trigger bugs, we model two general bug oracles, *fee consistency property (FCP)* and *resource consistency property (RCP)* (§4.3). FCP oracle requires identical gas charges that consume the same resource consumption. RCP oracle stipulates that equal resource consumption results in the same gas charges. Besides, Auspex also integrates differential testing [49, 58, 75]. Auspex will detect bugs in two chains with different configurations, which enhances bug detection capability in divergent contexts.

Evaluation. We implement and evaluate Auspex (§5 and §6). (i) To assess the capability of bug identification, Auspex is deployed in Ethereum and discovers 13 new unknown TFM inconsistency bugs (§6.1). (ii) To understand the efficacy, we compare Auspex with the state-of-the-art baseline (§6.2). The results demonstrate that Auspex achieves 3.5 times more code branch coverage. (iii) To analyze the core design of Auspex, we also conduct the ablation and contrast evaluation on three components of Auspex (§6.3).

Impact. We further explore the economic and security impact of TFM inconsistency bugs. (i) To understand the economic impact of these bugs, we select five bugs that can cause financial loss and quantify their economic losses across 1.5M blocks on Ethereum and BSC (§7.1). The results showed that these bugs caused economic losses exceeding one million dollars for users on each of the respective two blockchain platforms. (ii) To assess the security risk of these bugs, we construct a type of DoS attack exploiting two of these bugs and test the DoS attack on the Ethereum testnet (§7.2). We find that the DoS attack can increase the block size by 96 times and transaction wait time by 4.5 times in testnet.

Contributions. In summary, we make four contributions:

- We design the first tool, Auspex, to automatically detect the TFM inconsistency bug in Ethereum.
- Auspex involves three key technologies. (i) *A chain-based strategy* that considers constructing the test cases from four levels (chain configurations, blocks, transactions, and contracts), to generate high-quality fuzz input efficiently. (ii) *A charging-guided fuzzing approach* that utilizes the relations (between gas fee and its resource cost) and inconsistency (between two chains with different configurations) as the fuzzing feedback, to improve the coverage of blockchain's implementation. (iii) We propose two general oracles, *FCP and RCP*, to automatically detect TFM inconsistency bugs.
- We evaluate the efficacy of Auspex in Ethereum. Auspex detects 13 unknown TFM inconsistency bugs and discovers over 3.5 times more code branches than the advanced tool.
- We evaluate the financial and security impact of the TFM inconsistency bugs. We find the bugs have caused exceeding millions of dollars for users on Ethereum and BSC. Besides, we construct a DoS attack exploiting these bugs, which causes block sizes to increase by 96 times and the transaction waiting time to increase by 4.5 times in testnet.

2 Background

Ethereum. *Ethereum* [72] is the largest blockchain platform supporting smart contracts. *Account* [72] is the most basic entity in Ethereum, which has two types, *externally owned account (EOA)* [72] and *contract account (CA)* [72]. EOA interacts with Ethereum via *external transaction* [38] (termed as $tx_{external}$), which are categorized into three distinct types: transfer of funding to EOA, creation of a program (i.e., smart contract) on CA, and invocation of a program on CA. If a CA invokes another CA, we represent it as *internal transaction* [38] (termed as $tx_{internal}$). *Smart contracts* are the turing-completeness program that executes atop the Ethereum [72], which will be programmed at a high-level program language (e.g., *solidity* [20]) and subsequently compiled into low-level opcodes [75]. Those opcodes will be interpreted and executed by *Ethereum virtual machine (EVM)* [34, 72]. Besides, there are some specialized smart contracts, *precompiled contracts* [10, 45], that execute predefined native code within the EVM, offering complex cryptography computations. The precompiled contracts are deployed at the address `0x1` to `0xA`.

Ethereum adopts *state* [35, 50, 54] to persistently store the account information on disk, mainly encompassing balances, contract bytecodes, and contract variables. Concretely, Ethereum organizes the account information into a huge tree structure, *Merkle Patricia Trie (MPT)* [44, 69, 72]. All of the accounts (EOAs and CAs) will be stored in the leaf nodes of *state trie* [50], which stores balances and contract bytes. The leaf node of CA on state trie has a pointer to the *storage*

tries, which maintains an independent space to store the persistence variables of the contract. CAs access and modify the persistent variable by the EVM opcode `SLOAD` and `SSTORE`.

Transaction fee mechanism (TFM). *Transaction fee mechanism (TFM)* [52, 53, 72] is used to deter the abuse of on-chain resources (e.g., storage and computation) and incentive the miners/validators [41]. Specifically, TFM of Ethereum allocates on-chain resources through a pricing mechanism named *gas*, and different operations that utilize different resources require different amounts of gas. For example, opcode `SLOAD` needs 100 gas to load data from the memory and 2,100 gas to load data from disk [15, 72]. According to resource usage, every transaction could be charged with different gas consumption. The gas will be converted to *ether* by the *gas price* recording in the block header. The ether is the native cryptocurrency of Ethereum, and the gas price is set by the sender of the transaction [72].

In essence, gas costs are intended to accurately represent the resource utilization of on-chain operations, serving as a safeguard against resource abuse [72]. However, inconsistencies between gas costs and actual resource consumption compromise blockchain security by creating exploitable vulnerabilities, such as enabling spam transaction attacks, while also introducing inefficiencies into the system. Addressing these inconsistencies is critical to preserving the blockchain ecosystem's security and economic stability, as emphasized by both academic research [37, 61] and industry [15] best practices.

Next, we list the related gas fees of our work in Table 1 from Ethereum specification [72]. We elaborate on the notions of gas fee, its gas amount, and description as follows.

(i) $g_{\varnothing}$, $g_{verylow}$, and $g_{precompiled}$. $g_{\varnothing}$ represents the operations without gas fee, and $g_{verylow}$ of 3 gas is paid by the opcodes with low cost (e.g., opcode `PUSH`). The $g_{precompiled}$ is used to invoke a precompiled contract. The precompiled contracts perform different cryptographic calculations with different gas fees [10] (e.g., 3,000 gas for `ECRECOVER` and 600 gas for `RIPEMD-160`). Note that Ethereum specification does not define $g_{\varnothing}$ and $g_{precompiled}$, we introduce them to ensure the completeness of our work.

Table 1: The notions of gas fees and its details.

Name	# of gas	Description
$g_{\varnothing}$	0	Operations with no cost.
$g_{precompiled}$	N.A.	The cost of executing recompiled contracts.
$g_{verylow}$	3	Opcodes with low cost, e.g., opcode `PUSH`.
$g_{warmaccess}$	100	Accessing data in cache, e.g., opcode `SLOAD`.
$g_{coldaccountaccess}$	2,600	Accessing account in disk, e.g., opcode `BALANCE`.
$g_{coldsload}$	2,100	Accessing slot from disk, e.g., opcode `SLOAD`.
g_{sset}	20,000	Opcode `SSTORE` sets slot from zero to none-zero.
$g_{acladdress}$	2,400	Accessing an account with the *access list* [16].
$g_{aclstorage}$	1,900	Accessing a storage with the access list.
$g_{callvalue}$	9,000	Opcode `CALL` transfers non-zero value.
$g_{newaccount}$	25,000	Opcode `CALL` creates a new account.
g_{tx}	21,000	The basic cost for transactions. .
$g_{txdatazero}$	4	Per zero byte of inputdata in a transaction.
$g_{txdatanonzero}$	16	Per non-zero byte of inputdata in a transaction.
$g_{codedeposit}$	200	Byte for the bytecode of the created contract.

We list the related opcodes of these gas fees in Appendix A.

(ii) $g_{warmaccess}$, $g_{coldaccountaccess}$, $g_{coldsload}$, and g_{sset}. The four gas fees are mainly used to access and modify the blockchain state (i.e., account or variable), which is introduced in EIP-2929 [15]. When a transaction first accesses and modifies an account (resp. slot), it will be charged as $g_{coldaccountaccess}$ of 2,600 gas (resp. $g_{coldsload}$ of 2,100 gas). If the transaction continues to access or change the same state, it will be charged a much cheaper fee (i.e., $g_{warmaccess}$ of 100 gas) as the state data is already in the memory. g_{sset} is mainly used to modify a slot from zero to non-zero. The reason why g_{sset} (i.e., 20,000 gas) is much higher than the other three gas fees is that it will cost more storage resources. When a user sets a slot from zero to non-zero, Ethereum will insert this new slot into the storage trie, and cause more write workloads of disk.

(iii) $g_{acladdress}$ and $g_{aclstorage}$. The two gas fees are paid for the *access list*, which is introduced in EIP-2930 [16]. The sender first predefines the accounts and slots (i.e., contract variables) into the access list in the block header, which obtains cheaper gas fees to access them, e.g., $g_{acladdress}$ of 2,400 gas is lower than $g_{coldaccountaccess}$ of 2,600 gas.

(iv) $g_{callvalue}$ and $g_{newaccount}$. The two gas fees are mainly used to transfer ether for opcode `CALL`. When the sender transfers some ether, it will be charged as $g_{callvalue}$ of 9,000 gas. If the receiver account does not exist before, the sender should pay a more $g_{newaccount}$ of 25,000 gas as it needs more storage resources to insert the new account into account trie.

(v) g_{tx}, $g_{txdatazero}$, $g_{txdatanonzero}$, and $g_{codedeposit}$. The four gas fees are adopted to send a transaction. The sender must pay g_{tx} of 21,000 gas to launch a transaction, which contains 9,000 gas for two accounts writing, 3,000 gas for signature verification, 6,800 gas for transaction data, and 2,200 gas for transaction-specific overhead [36]. If the transaction has additional inputdata, the sender will be charged as $g_{txdatanonzero}$ of 16 gas (resp. $g_{txdatazero}$ of 4 gas) to pay for per non-zero (resp. zero) byte in the inputdata. While the transaction is used to deploy the contract, the sender should pay for the deployed contract with $g_{codedeposit}$ of 200 gas per byte for bytecode.

3 Motivation

System model. We adopt the following system model in our work. First, we formally define the blockchain system as $\Upsilon = \{B, A, R, G\}$, which represents blocks, accounts, resources, and gas fees. Specifically, finite set $B = \{b_1, b_2, \ldots, b_n\}$ denotes the blockchain of Ethereum. Each block $b_i \in B$ contains a series of transaction $\{tx_1, tx_2, \ldots, tx_n\}$. Then, finite set $A = \{a_1, a_2, \ldots, a_n\}$ denotes the account of blocks. We use finite set $G = \{g_{\varnothing}, g_{precompiled}, \ldots, g_{codedeposit}\}$ to represent the gas fee sets, which was defined in Table 1.

Next, we model and profile the on-chain resource consumption by memory write/read, disk write/read, CPU compute, network transfer, and zero cost. We define them as $R =$

$\{r^l_{memw}, r^l_{memr}, r^l_{diskw}, r^l_{diskr}, r^l_{cpu}, r^l_{net}, r^0_\varnothing\}$. Concretely, the notion's superscript represents the data length, and the notion's subscript represents the type of resource consumption. Then, we adopt the symbol $\rightarrow$ to represent the *charging relation* of gas fee and resource consumption, e.g., $g_{precompiled} \rightarrow r^l_{cpu}$ denotes the computing resources charged by the gas fee of the precompiled contract. Note that one gas fee may be charged for the consumption of multiple resources. For instance, the gas fee of opcode `EXTCODESIZE` will be charged for two disk access (i.e., $g_{coldaccountaccess} \rightarrow (r^{len(account)}_{diskr}, r^{len(code)}_{diskr})$), which represents `EXTCODESIZE` first reads account from the disk, and then reads bytecode size based on account from disk. Finally, we use the symbol $<$ to represent the comparison relationship between different resources and gas fees. To handle the multiple resources, we further extend the four rules to the comparison relationship (i.e., symbol $<$) based on resource number, type, and data length. (i) For resources of the same type, those with fewer usage occurrences are considered smaller than those with more frequent usage, e.g., $r^{len(account)}_{diskr} < (r^{len(account)}_{diskr}, r^{len(code)}_{diskr})$. (ii) For resources with the same data length, memory-based consumption is considered smaller than disk-based consumption, e.g., $r^l_{memr} < r^l_{diskr}$. (iii) With the same type of consumption, resource consumption with the shorter data length is smaller than that with longer data length, e.g., $r^{16Bytes}_{diskr} < r^{32Bytes}_{diskr}$. (iv) In the context of gas fees, the symbol $<$ still indicates the comparison amount of gas fee. Then, we define two bug types.

Definition 1: Resource inconsistency bug (RIB). *RIB* means the blockchain provides different utilization of resources under the same gas fee. Specifically, RIB occurs when the blockchain overcharges the gas fee with the actual resource the user consumes. This means that the resources consumed by a user's transaction or operation on the blockchain are lower than expected based on the gas fee paid. Such RIBs will cause the user financial loss because of overcharging and potentially have significant implications for the fairness and trust of the blockchain system. We formally define the rules of RIB, $RIB \models (g_i \rightarrow r^l_j) \wedge (g_i \rightarrow r^l_k) \wedge (r^l_j < r^l_k)$, where $g_i \in G$, and $r^l_j, r^l_k \in R$. The rule denotes that the gas fee of g_i pays different resources r^l_j and r^l_k, and the resource cost of r^l_j is lower than r^l_k. Note that we will use the representation that the gas fee corresponds to only one resource consumption in the rest of the definitions because of the page limitation. To better understand RIB, we interpret two RIBs, bug#3 and bug#4, which are uncovered by Auspex.

(i) Bug#3. The bug#3 lies in the fact that a disk write operation is not executed, yet a gas fee is still charged. Specifically, when the sender of the transaction is identical to the receiver, the ether transfer does not occur, and the balances of the two accounts aren't updated. But the user still will be charged with $g_{callvalue}$ (or 9,000 gas in g_{tx}). There are three scenarios of bug#3, including $tx_{external}$, opcode `CALL` is invoked to transfer ether, opcode `CALLCODE` is invoked to transfer ether. Note that the semantic [47, 72] of opcode `CALLCODE` is to transfer ether on an account itself. Hence once the opcode `CALLCODE` is invoked for ether transferring, bug#3 will be triggered. The bug#3 holds in RIB, as $RIB_{bug3} \models (g \rightarrow (r^{len(account)}_{diskw}, r^{len(account)}_{diskw})) \wedge (g \rightarrow r^0_\varnothing) \wedge (r^0_\varnothing < (r^{len(account)}_{diskw}, r^{len(account)}_{diskw}))$, where $g \in \{g_{callvalue}, g_{tx}\}$. Note that for $tx_{external}$, the account of the sender will still be updated for adding nonce. However, this situation still meets the definition of RIB, when we replace the $r^0_\varnothing$ as $r^{len(account)}_{diskw}$.

(ii) Bug#4. Bug#4 is triggered because the reverted transaction does not update the state of the blockchain in the disk but is still overcharged for the gas fee of disk writing. Concretely, whenever a normal transaction updates the state (e.g., modifying the balance of an account), the updated state is not immediately written to disk but is temporarily stored in memory (i.e., cache) until all transactions in the block are completely executed, at this moment the corresponding changed state is updated into disk. However, if the transaction is reverted, the updated state will be reset (i.e., roll back) in memory and will not update into disk. In summary, the normal transaction and reverted transaction pay the same gas fee to update the state, but the normal transaction writes the update into the disk while the reverted transaction only updates into memory. The bug#4 affects all the gas fees of state updating, encompassing $g_{callvalue}$, $g_{newaccount}$, $g_{coldsload}$, g_{tx}, and $g_{codedeposit}$. Note that bug#4 is also used in the case where a contract invoking is reverted when modifying a slot. The bug#4 holds in RIB, as $RIB_{bug4} \models (g \rightarrow r^{len(account)}_{diskw}) \wedge (g \rightarrow r^{len(account)}_{memw}) \wedge (r^{len(account)}_{memw} < r^{len(account)}_{diskw})$, where $g \in \{g_{callvalue}, g_{newaccount}, g_{coldsload}, g_{tx}, g_{codedeposit}\}$.

Definition 2: Fee inconsistency bug (FIB). *FIB* denotes the user pays different fees for consuming the same on-chain resources. Concretely, FIB manifests when the TFM fails to consistently apply uniform charges for identical resource consumption. Firstly, FIB can lead to scenarios where certain users can exploit the fee mechanism, resulting in optimized charges for their on-chain activities. Secondly, FIB leads to insufficient gas fees that can be exploited by DoS attacks, which potentially degrades the performance and reliability of the blockchain. Formally, FIB follows the rule, $FIB \models (g_i \rightarrow r^l_j) \wedge (g_k \rightarrow r^l_j) \wedge (g_i < g_k)$, where g_i and $g_k \in G$, and $r^l_j \in R$. The rule denotes that the gas fee of g_i and g_k pays for the identical resources r^l_j, but the gas fee of g_i is lower than g_k.

To further understand FIB, we take bug#13 as an example. The bug#13 arises $tx_{internal}$ should pay the gas fee of $g_{newaccount}$ to write a new account into a disk, whereas $tx_{external}$ do not (i.e., $g_\varnothing$). The new account means an account that previously did not exist in the state. Concretely, when launching $tx_{external}$ transferring ether to a new account, the user does not charge any fee to update the new account into the disk (i.e., state trie). While invoking the contract to transfer ether to a new account, the user will be charged $g_{newaccount}$ to update the new account into disk. The bug#13 holds in FIB as $FIB_{bug13} \models (g_{newaccount} \rightarrow r^{len(account)}_{diskw}) \wedge (g_\varnothing \rightarrow$

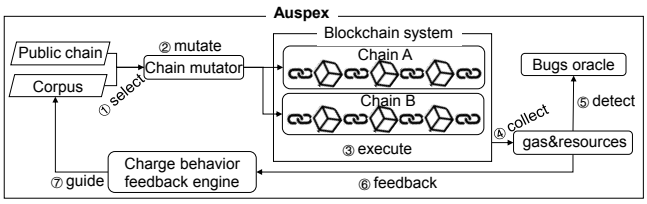


Figure 1: The overview of Auspex. ① Auspex constructs complete chains as test cases from the corpus. ② The complete chains will be mutated in four levels. ③ Auspex will execute two chains with different configurations in parallel. ④ The code coverage and charge relations will be collected by Auspex. ⑤ The bug oracles detect the TFM inconsistency bug in real time based on the information of charging relation. ⑥ The code coverage, charging relation, and inconsistency will be feedback to guide the fuzz process. ⑦ Any complete chain that improves code coverage or uncovers new charging relationships will be added to the corpus.

$r_{diskw}^{len(account)}) \wedge (g_{\varnothing} < g_{newaccount})$.

Utilizing bug#13, users can save 4,000 gas if they transfer ether to a new address through $tx_{external}$ before transferring ether to the new address through the contract. Concretely, if the user needs to use the contract to transfer ether to some new accounts (e.g., airdrop), the user can first pay g_{tx} of 21,000 gas to send some $tx_{external}$ to transfer 1Wei (10^{-18} ether) to those new accounts. After that, the user will not be charged with $g_{newaccount}$ of 25,000 gas when the contract transfers ether to the accounts, which have existed in the state. Therefore, bug#13 makes the user save 4,000 gas, which is obtained by the difference between $g_{newaccount}$ of 25,000 gas and $tx_{external}$ of 21,000 gas. Undoubtedly, if the users directly use $tx_{external}$ to transfer ether to the new address, they can save more gas. Besides, we exploit the FIBs to construct a DoS attack in §7.2.

4 Auspex Design

Auspex is designed to detect TFM inconsistency bugs in Ethereum. Specifically, it aims for the following three *design goals*. (i) *Efficiency.* Utilizing the chain-based strategy, Auspex can efficiently construct complete chains with varied configurations as test inputs (§4.1). (ii) *Adequacy.* Auspex adopts the charge-guided fuzzing approach to trigger more inconsistency between the gas fee and the cost of resources as much as possible (§4.2). (iii) *Automation.* Auspex deploys two oracles to monitor the blockchain in real-time, and automatically uncover TFM inconsistency bugs (§4.3).

Fig. 1 illustrates the overview of Auspex. ① Auspex constructs a complete chain as a test case from the corpus or the public blockchain. ② The test case (i.e., complete chain) will be mutated from four levels, which encompasses chain configurations, blocks, transactions, and contracts. ③ Two complete chains with different configurations will be executed in the real blockchain system in parallel. ④ In the meanwhile, Auspex will monitor the blockchain system to collect the data of code coverage and the charging relations (i.e., gas fees and on-chain resources) in real-time. ⑤ The bug oracle identifies bugs by analyzing gas charging relationships. It detects bugs not only within individual chains but also across chains with different configurations, enabling comprehensive analysis. ⑥ Auspex takes code coverage, charging relation, and inconsistency between two chains as the feedback to guide the fuzz process. ⑦ Auspex will put the complete chains into the corpus if the chains can update the code coverage and trigger new charging relations. The above seven steps will be executed and iterated as the fuzz loop until termination.

4.1 Test case generation

Test case. Auspex adopts the chain-based strategy (CBS) to generate and mutate the test cases. Specifically, CBS defines a complete chain as the test cases with four levels as presented in Fig. 2. The data structure of test cases encompasses chain configuration, block field, transaction context, and contract operation. The data structure of test cases includes three key features for efficient fuzzing of Auspex.

(i) Auspex enables mutating the configuration of the chain, which indicates the launching parameters within the chain implementation (e.g., geth) that execute the blocks. Previous works [39, 75] only generate the test cases from blocks, transactions, and contracts. They execute the chain with default launching parameters, which can't adequately trigger the behavior of resource consumption. While Auspex generates and tests the chain execution with different launching parameters to trigger more behaviors of resource consumption. Specifically, through the analysis of Ethereum's specifications [72] and implementation [27], we identified three types of launching parameters related to resource consumption: cache configurations, database schema configurations, and fork configurations (see Appendix C). For example, by tuning the size of the cache (i.e., cache configurations), Auspex can unveil some bugs caused by disk and memory consumption.

(ii) We introduce opcode fields and data fields for the con-

```
type ChainConf struct {  //chain configurations
    CacheConf []int  //cache size
    DBConf int  //DB schema
    ForkConf int  //hard-fork version
    Blocks []Blockfield}
type Blockfield struct { //block fields
    Coinbase address //miner address
    ExtraData []byte //extra data
    Txs []TxContext}
type TxContext struct { //transaction contexts
    Gas uint64 // gas cost
    Value uint256 //Ether value
    To address // receiver address
    Accesslist []address //access list
    Inputdata []byte }//inputdata
type ContractOps struct { //contract operations
    Opcode []byte //opcodes
    Operands []byte }//operands
```

Figure 2: The data structure of test cases applied by Auspex.

tract. Concretely, we decouple the bytecode of the contract with the opcodes (i.e., opcode fields) and its operands (i.e., data fields). For different fields, we apply the adaptive mutation on different fields to preserve the semantics of the contract bytecode. For opcode fields, we consider the operational semantic [46, 47] to mutate it by changing opcodes into alternatives that necessitate similar contexts, such as the number of stack operations, memory operations, and state operations. For the data field, we adopt random mutation.

(iii) We limit the range of parameters that have minimal impact on transaction execution. Such limitations for the parameters avoid the time overhead of testing irrelevant variations. For instance, Auspex will keep the fields like `ExtraData` and `Coinbase` in `Blockfield` in a fixed range, while mutating the fields like `Gas` and `Value` in `TxContext` more frequently. This approach focuses computational resources on impactful mutations, optimizing the fuzzy's efficiency.

Mutation. To generate high-quality test cases, Auspex first keeps the type information of the fields to ensure syntactic correctness. Concretely, there are three field types in the test case, containing integer, integer array, and bytes array. Auspex will retain the type of fields during the mutation. Then, to ensure semantic integrity, Auspex applies an adaptive mutation strategy based on the fields' functionality in block execution.

(i) `ChainConf`. Auspex first adopts random mutate strategy on the fields of `CacheConf`, `DBConf`, and `ForkConf`. Then, Auspex applies operations of copy, add, and delete to the block list (`Blocks`). When Auspex generates a `ChainConf` for a complete chain, the chain will keep one `ChainConf` unchanged during the current fuzzy iteration.

(ii) `Blockfield`. Auspex applies the operation of randomly changing in `Coinbase` and `ExtraData` fields. Note that we will randomly select some values from a set of fixed values, as these two fields have no impact on the block execution. For `Txs` fields, Auspex will conduct copy, add, and delete operations on the transactions list in a block.

(iii) `TxContext`. Auspex will randomly mutate `Gas` and `Value` from minimal value to their threshold. Then, Auspex utilizes a semantic dependency strategy to mutate the three fields (i.e., `to`, `Accesslist`, and `Inputdata`), as the value of `to` will affect `Accesslist` and `Inputdata`. If `to` is mutated to an EOA, Auspex will randomly select an EOA from a predefined set of addresses and set the `Accesslist` with `Inputdata` to nil. If `to` is mutated to a CA, Auspex will randomly select a contract address from a set of existing CAs in the blockchain state. For `Inputdata`, we will parse the bytecode of the new CA to get the function signatures of 4 bytes (cf. Appendix D) and mutate the first 4 bytes of `Inputdata` as one of the valid function signatures. For the rest slice of `Inputdata`, we use the byte array mutation operation, including insertion, deletion, and replacement. For `Accesslist`, after mutating address `to`, it is hard to predict the accounts and storage slots that transactions will access. Therefore, instead of specifying the account and storage slot directly, we generate a sequence of numbers, each number n representing the nth account and storage slot accessed by the transaction. Hence, we take the `Accesslist` as an integer array to conduct the mutate operation of insertion, deletion, and replacement.

(iv) `ContractOps`. The mutation process of `ContractOps` is categorized into syntax and semantics. Regarding syntax, any modification to an opcode is accompanied by corresponding adjustments to the stack operations to ensure stack balance. Regarding semantics, Auspex prioritizes mutating opcodes to those with similar semantics. This ensures that the modified opcodes perform equivalent stack operations, memory operations, storage operations, control flow transfers, and context accesses. Besides, we do not mutate the function selector in bytecode to ensure that the contract executes properly. Auspex adopts a strategy for mutating `opcode` using add, delete, copy, and change operations. For the `operands`, Auspex introduces random mutation on immediate values, stack values, memory addresses, and storage slots.

Algorithm 1: charging-guided fuzzing approach

Input: *BS*: Blockchain system
Output: *Bugs*: TFM inconsistency bug

```
1  BS.Deploy()
2  Corpus.Init(Mainnet)
3  Bugs ← {}
4  while True do
5      Chain' = Select(Corpus)
6      Chain, conf1, conf2 = Mutation(Chain')
7      for i = 1 to len(Chain) do
8          for j = 1 to len(block_i) do
9              Set^1_{g,g→r},Cov^1 = BS.Execute(tx_j,conf1)
10             Set^2_{g,g→r},Cov^2 = BS.Execute(tx_j,conf2)
11             async:
12                 newBugs=BugOracle(Set^1_{g,g→r},Set^2_{g,r,g→r})
13                 Bugs.append(newBugs)
14                 newcov=Coverage(Cov^1,Cov^2)
15                 diff=Inconsistency(Set^1_{g,g→r},Set^2_{g,r,g→r})
16                 if (newBugs >0) or (diff > 0) or
                    (newcov > 0) then
17                     Corpus.append(tx_j,conf1,conf2)
```

4.2 Feedback mechanism

To effectively explore TFM's code logic, we propose charging-guided fuzzing approach by considering two new feedback, charging relations and inconsistency on different chains.

Algorithm 1 provides the details of our approach. First, Auspex deploys the environment of block execution and initializes the seeds in Corpus from the mainnet (Line 1-3). In each iteration, Auspex will select the original chain from the Corpus and mutate the original chain to the new chain as a test case (Line 5-6). Then, the blockchain system will execute the chain of test cases with different chain configurations (Line 7-10). Meanwhile, Auspex will collect information on gas fees,

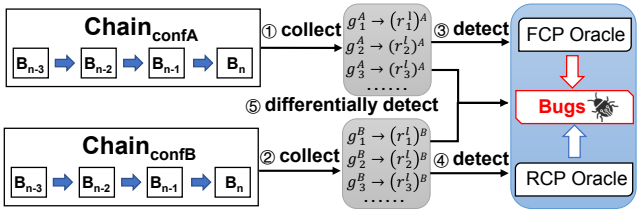


Figure 3: The bug detection of Auspex. ①&② Auspex collects the charging relations independently from two blockchains with different configurations. ③&④ Auspex conducts a separate detection on the charging relation in each chain, comparing the latest charging relation with the previously collected relations. ⑤ Auspex will apply differential bug detections between two executed blockchains.

charging relations, and code coverage. Next, the bug oracles will analyze the gas fees and charging relation to detect TFM inconsistency bugs (Line 12-13), which will be explained in §4.3. Besides, Auspex will check the new inconsistency (we will explain later) and new code coverage from the collected information (Line 14-15). The new inconsistency, charging relations, code coverage, and bug detection will be the fuzz feedback, and any test cases that trigger them will be stored in the corpus (Line 16-17). Next, we interpret the feedback mechanism of Auspex as follows.

(i) Code coverage and bug detection. Auspex adopts code coverage and bug detection as feedback whenever a test case triggers some new codes or unveils some new bugs. Auspex will preserve those test cases in the corpus, which facilitates exploring more implementation for the blockchain system.

(ii) Gas fees and charging relations. Auspex applies the gas fees and charging relations as novel fuzz feedback. For gas fees, we consider the combinations of the gas types as the fuzz feedback. Concretely, Ethereum specification [72] defines 39 types of gas fees, which means we have a total of 2^{39} combinations. As for the charging relations, we adopt the same combination method and consider combinations on both gas types and resource types (i.e., G and R in §3). If the test case triggers new combinations, Auspex will save it in the corpus. This new feedback enables Auspex to cover more scenarios of gas and resource consumption.

(iii) Inconsistency. Auspex takes inconsistency between the two chains with different configurations as feedback. Auspex will send feedback when the same transaction has different gas fees, resource consumption, and code coverage when executed on two chains. Formally, we define the `Inconsistency()` as $diff(g_i^1, g_i^2) + diff((r_i^l)^1, (r_i^l)^2) + diff(Cov_i^1, Cov_i^2)$. This feedback enables Auspex to preserve the test cases which can trigger inconsistency of TFM as much as possible.

In the seed selection strategy in Algorithm 1, seeds are prioritized based on effectiveness in identifying bugs, triggering new charging behaviors, uncovering novel gas fee combinations, and detecting inconsistencies in two chains. When no new behaviors are observed, the strategy defaults to prioritizing seeds that can discover new code coverage, ensuring thorough exploration of previously untested code paths.

4.3 Bug detection

We model two bug oracles to uncover TFM inconsistency bugs and formally represent them as follows.

Definition 3: Fee consistency property (FCP). FCP asserts that for any given type of gas fee, the corresponding resource consumption must remain consistent. Formally, FCP can be represented as $FCP \models (g_i \rightarrow r_j^l) \wedge (g_i \rightarrow r_k^l) \wedge (r_j^l = r_k^l)$, where $g_i \in G$, and $r_j^l, r_k^l \in R$. FCP ensures that transactions with identical gas fee types incur equivalent resource usage (e.g., computation and storage), thereby maintaining fair resource allocation within the blockchain system.

Definition 4: Resource consistency property (RCP). RCP stipulates that for any given type of resource consumption, the corresponding gas fee must be uniform. We formally define RCP as $RCP \models (g_i \rightarrow r_k^l) \wedge (g_j \rightarrow r_k^l) \wedge (g_i = g_j)$, where $g_i, g_j \in G$, and $r_k^l \in R$. RCP ensures that similar resource usage incurs the same gas fee, providing a stable fee structure across all transactions in the blockchain network.

Oracle detection. Auspex deploys the oracles with FCP and RCP to monitor the blockchain executions by combining the single-target test and differential fuzz test. Fig. 3 visualizes the process of bug detection via oracles. ①&②: Auspex initiates the process by independently collecting charging relations from two blockchains, which ensures a diverse set of charging relations can reflect various blockchain configurations. ③&④: Auspex performs an intra-chain detection by comparing the latest charging relations within each blockchain against the previous charging relations. This comparison identifies any inconsistency bugs in gas fees and resource consumption that may indicate a violation or adherence to FCP and RCP within a single blockchain configuration. ⑤: Auspex applies differential detection between the two executed blockchains. By comparing the charging relations across different blockchain configurations, Auspex identifies discrepancies that may indicate violations of FCP and RCP across varying blockchain configurations.

After FCP and RCP oracles are triggered, we adopt the following procedure to locate the root cause. First, we reproduce the bugs using historical test cases to manually analyze charging relationships (e.g., gas costs and resource utilization). Then, we conduct a thorough audit of the TFM implementation to inspect FIB and evaluate resource components to uncover RIB. Once the root causes are identified, we validate with official blockchain specifications, ensuring that the detected bugs align with protocol definitions.

5 Implementation

Auspex aims to detect TFM inconsistency bugs via fuzz technology. The main components of Auspex include a muta-

Table 2: Descriptions for the 13 unknown TFM inconsistency bugs identified by tools

#	Type	Bug Description
1	**FIB**	Invoking precompiled contracts does not pay the gas fee of inputdata (i.e., $g_{txdatazero}$ or $g_{txdatanonzero}$).
2	**FIB**	The gas fee of inputdata (i.e., $g_{txdatazero}$ or $g_{txdatanonzero}$) and bytecode (i.e., $g_{codedeposit}$) are inconsistent in writing data in disk.
3	**RIB**	Transfers to oneself do not update balance but charge gas fees (i.e., $g_{callvalue}$ and g_{tx}).
4	**RIB**	The reverted transaction only updates state in memory but is charged as disk writes (i.e., $g_{callvalue}$, $g_{newaccount}$, $g_{coldsload}$, g_{tx}, and $g_{codedeposit}$).
5	**RIB**	In-transaction accesses to the same account are memory-based after the first access but are charged as disk access (i.e., $g_{callvalue}$).
6	**FIB**	Miner/validator rewards(or withdraws [22]) update the balance of state in the disk but no gas fee is charged (i.e., $g_{\varnothing}$).
7	**FIB**	The disk read of `SSTORE` opcode is not charged (i.e., $g_{\varnothing}$).
8	**FIB**	The gas fee of `SLOAD` (i.e., $g_{warmaccess}$) and `MLOAD` (i.e., $g_{verylow}$) are inconsistent in memory reads.
9	**RIB**	The state reading is optimized, yet gas fees (i.e., $g_{coldaccountaccess}$, $g_{coldsload}$, $g_{acladdress}$, and $g_{aclstorage}$) remain unaltered.
10	**FIB**	$tx_{internal}$ updating accounts in disk does not charge gas fees (i.e., $g_{\varnothing}$).
11	**RIB**	`Ext*` opcodes (e.g., `EXTCODESIEZE`), accessing extra bytecode data over opcode `BALANCE`, but charge an identical gas fee of $g_{coldaccountaccess}$.
12	**RIB**	User charges for slots on their access list (i.e., $g_{acladdress}$ and $g_{aclstorage}$), even if those slots are not accessed.
13	**RIB**	$tx_{internal}$ will be charged the gas fee for updating a new account in state (i.e., $g_{newaccount}$), yet $tx_{external}$ will not.

tion engine, blockchain execution environment, and feedback mechanism. **(i) Mutation engine.** We implement the mutation engine of Auspex based on go-fuzz [26]. Mutations include random charges, boundary value testing, and semantic modifications to ensure a wide range of test cases. **(ii) Blockchain execution environment.** We employ geth as the blockchain execution environment to ensure its compatibility with our requirements. We modify the implementation of the geth client to execute the blockchain generated by Auspex. The fuzzed test cases are executed in two geth instances with the chains containing 1,000 blocks. **(iii) Feedback mechanism.** The feedback mechanism in Auspex guides the fuzzing process by taking code coverage, charging relations, and inconsistency as feedback. Using `go-fuzz`, we monitor which parts of the code are exercised by each test case, allowing Auspex to prioritize test cases that increase coverage. Concretely, we mainly focus on the code coverage of transaction execution and charging components, such as EVM, TFM, chain executor, and state storage. Such scope ensures that all critical aspects of the TFM and its resource utilization are thoroughly analyzed. Besides, we modify the code on geth to collect gas fees and resource consumption in geth, which also applies to prioritize test cases and bug detection.

6 Evaluation

We evaluate Auspex with the following research questions.
RQ1: Can Auspex identify TFM inconsistency bugs? (§6.1)
RQ2: Can Auspex outperform baselines in terms of both bug-finding capability and coverage? (§6.2)
RQ3: How does Auspex's bug-finding performance benefit from its design? (§6.3)

6.1 RQ1: Bug-finding capability

We deploy Auspex on Ethereum for 24 hours to detect TFM inconsistency bugs on a server featuring the Intel Xeon Gold 5218R CPU (2.10 GHz, 10 cores), 64 GB of RAM, and a 4TB Samsung 860 EVO SSD. Ultimately, Auspex successfully identified a total of 13 previously unknown TFM inconsistency bugs, which encompasses 6 FIBs and 7 RIBs. Table 2 presents all 13 bugs. We will interpret bug#1,2,5,9 as they will be utilized in §7.1 and §7.2. The rest of the bugs will be illustrated in Appendix B as the page limitation.

Bug#1. The bug#1 is caused by users using the computing resources of precompiled contracts with different gas fees. Specifically, precompiled contracts need to pay the gas fee for specifying parameters to execute them. However, when invoking the precompiled contract `ECRECOVER`, the user does not need to pay for specifying parameters as Ethereum will pad the parameters with all zero to execute `ECRECOVER`. For example, transaction `0x1fb0c` [1] paying 24,000 gas (= $gas_{tx} + gas_{precomplied}$) and transaction `0x6b016` [2] paying 24,276 gas (= $gas_{tx} + gas_{precomplied} + gas_{txdatanonzero}$) invokes precompiled contract `ECRECOVER`. Formally, as $FIB_{bug1} \models (g_1 \rightarrow r_{cpu}^{128B}) \wedge (g_2 \rightarrow r_{cpu}^{128B}) \wedge (g_1 < g_2)$.

Furthermore, bug#1 resulted in unforeseen economic losses for users. The user mistakenly considers the precompiled contract addresses (from `0x1` to `0xA`) as a burn address [38], and the user transfers cryptocurrency to a precompiled contract address. Such a mistake will cause two unexpected results. (i) The user considers the precompiled contract address as EOA, and sets 21,000 (i.e., gas_{tx}) as the gas limit for their transaction. The transaction fails to execute as the gas limit is not enough to pay the gas fee of $gas_{tx} + gas_{precomplied}$. Finally, the user not only lost all the gas fees but also the transaction will be reverted, e.g., transaction `0xe5037c` [3]. (ii) The user set a higher gas limit that just exceeded the gas fee of $gas_{tx} + gas_{precomplied}$ and successfully executed the transaction. Although the transaction is successfully executed, the user overpaid the gas fee of $gas_{precomplied}$, e.g., transaction `0x1fb0c` [1]. The rationale for these consequences lies in the unique addresses (i.e., `0x1` to `0xA`) of precompiled contracts, which are often misconstrued as burn addresses by users, including popular blockchain explorers [8] (see Fig. 4).

Bug#2. Bug#2 arises due to the disparity in gas fees, where the gas fee for writing `inputdata` to disk is lower than that for writing bytecode, despite both operations consuming equivalent disk resources. The inputdata of the transaction

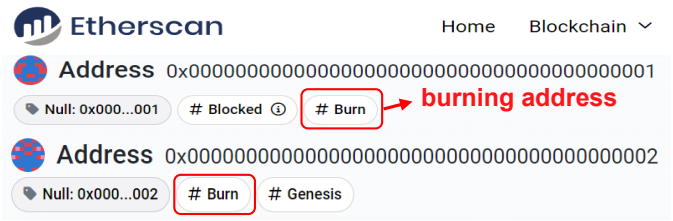


Figure 4: The blockchain explorer (e.g., Ethersacn) misleads the user to consider the precompiled address as the burning address. Such misleading causes financial loss for users.

will be stored in the block and written into the disk, which the user needs to pay $g_{txdatanonzero}$ of 16 gas (or $g_{txdatazero}$ of 4 gas) per byte. Meanwhile, when a transaction deploys a contract, the corresponding bytecode is also written to disk, with the user incurring $g_{codedeposit}$ of 200 gas per byte. Therefore, the inconsistency gas fees between inputdata and bytecode incur bug#2, which can formally be represented as $FIB_{bug2} \models (g_{txdatanonzero} \rightarrow r^{IB}_{diskw}) \wedge (g_{codedeposit} \rightarrow r^{IB}_{diskw}) \wedge (g_{txdatanonzero} < g_{codedeposit})$.

Bug#5. Bug#5 arises because when modifying an account, memory writes are erroneously charged at disk writes. Specifically, when a transaction updates an account's balance, the account will be loaded into the memory. Subsequent modifications to the balance of the same account should involve only memory writes, yet Ethereum charges these memory operations as the disk write. Formally, the bug#5 can be interpreted as $RIB_{bug5} \models (g_{callvalue} \rightarrow r^{len(account)}_{memw}) \wedge (g_{callvalue} \rightarrow r^{len(account)}_{diskw}) \wedge (r^{len(account)}_{memw} < r^{len(account)}_{diskw})$

Bug#9. The bug#9 is caused by the fact that Ethereum optimized the storage structure and reduced the reading of disk resources, but there was no corresponding gas fee modification. Concretely, Ethereum introduces a new structure, snapshot accelerate structure (SAS) [19, 32], which replaces MPT to accelerate disk reading. SAS organizes all state data as a flat key-value store, enabling reading data directly without reading the intermediate nodes of MPT [50]. Note that alongside geth, other Ethereum clients [18, 24, 29] also deploy similar accelerated reading structures [6, 9, 12], and we use SAS to denote this type of structure uniformly. However, after the storage resources consumed are optimized by SAS, the gas fees do not decrease accordingly, leading to bug#9. The bug#9 can be presented as $RIB_{bug9} \models (g \rightarrow r^{len(account)}_{diskr}) \wedge (g \rightarrow (r^{len(account)}_{diskr}, r^{len(MPT)}_{diskr})) \wedge (r^{len(account)}_{diskr} < (r^{len(account)}_{diskr}, r^{len(MPT)}_{diskr}))$, where $g \in \{g_{acladdress}, g_{coldsload}, g_{aclstorage}, g_{coldaccountaccess}\}$.

Answer to RQ1: *Auspex identifies 13 TFM inconsistency bugs on Ethereum platform, including 6 FIBs and 7 RIBs, significantly showcasing its capability in detecting such bugs.*

6.2 RQ2: Can Auspex outperform baselines?

This section demonstrates the efficacy of Auspex by comparing its code coverage and bug identification capabilities with state-of-the-art tools. We adopt the same hardware as §6.1.

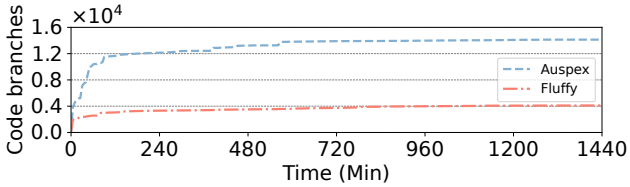


Figure 5: The code coverage of Auspex. We adopt Fluffy [75] as the baseline, and Auspex outperforms the baseline in the exploration of branch coverages at all times.

We first examine the code coverage of Auspex. To collect the information on code coverage, we launch Auspex for about 24 hours (i.e., 1,440 Minutes). Next, we select Fluffy [75] as the baseline because it represents the state-of-the-art fuzzer for Ethereum clients, embodying the most advanced techniques in fuzz testing. We run Fluffy under the same conditions to obtain coverage information. Fig. 5 visualizes the code coverages of branches for Auspex and Fluffy. Auspex (blue line) rapidly discovers a large number of code branches within the first 120 minutes, showing a steep growth, while Fluffy (red line) discovers significantly fewer branches during the same period. By the end of the experiment, Auspex explores nearly 1.4×10^4 branches, while Fluffy only covers roughly 4×10^3 branches by the end. This is because Auspex can detect more code logic in TFM and its resource utilization. It indicates that Auspex can explore more than 3.5 times code logic on Fluffy, and achieve significant fuzz performance.

Then, we examine the bug identification capability by comparing Auspex and Fluffy-FR. The Fluffy-FR is the baseline, which we equip the oracles of Auspex on Fluffy. We run the two tools for about 24 hours, and Fig. 6 provides the experiment results. Both tools, Auspex and Fluffy-FR, show rapid initial bug discovery in the first 240 minutes, but Auspex consistently detects more bugs at each time interval. Between 240 and 960 minutes, Auspex steadily finds additional bugs, ultimately identifying 13 bugs, while Fluffy-FR stagnates with no discoveries beyond 8 bugs. The Fluffy-FR doesn't find bug#1, bug#7, bug#9, bug#12, and bug#13. We take bug#9 as an instance, Fluffy-FR does not consider the mutation of chain configurations, and can not find the resource utilization difference between SAS and MPT in reading state. In summary, the differences in bug discovery suggest that Auspex

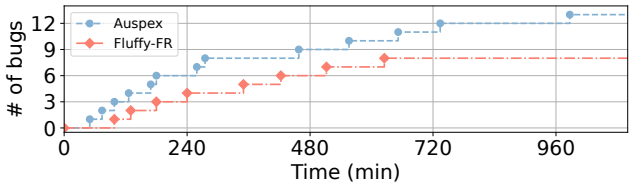


Figure 6: The bug identification capability of Auspex and baseline (i.e., Fluffy-FR, Fluffy [75] with FCP and RCP oracle). Auspex identifies 5 additional bugs than the baseline.

Table 3: The effect of four levels' mutation

L1	L2	L3	L4	Code Coverage	Identified Bugs
○	●	○	○	3,249	bug#6
○	○	●	○	4,651	bug#2,3,12
○	○	○	●	5,264	bug#4,5,7,8
●	●	○	○	3,632	bug#6,9
●	○	●	○	7,653	bug#1-3,9,12
●	○	○	●	9,163	bug#1,4,5,7,8-11
○	●	●	○	7,137	bug#2,3,6,12
○	●	○	●	6,452	bug#4-8
○	○	●	●	11,157	bug#2-5,7,8,10-13
●	●	●	○	9,613	bug#2,3,6,9-13
●	●	○	●	10,732	bug#1,4-11,13
●	○	●	●	13,498	bug#1-5,7-13
○	●	●	●	12,475	bug#2-8,10-13
●	●	●	●	14,130	bug#1-13

L1: ChainConf; L2: Blockfield; L3: TxContext; L4: ContractOps.

has a higher exploration capability and can uncover more complex bugs compared to Fluffy-FR.

Answer to RQ2: *Compared with the state-of-the-art tools, Auspex can cover more than 3.5 times branches, and uncover additional 5 bugs. It highlights Auspex's effectiveness in broader exploration and deeper identification.*

6.3 RQ3: Effects of components in Auspex

To obtain the effect of Auspex's core design in §4, we conduct some experiments on various components in Auspex, including mutation strategy, feedback mechanism, and differential detection. We adopt the same experiment specs as §6.1.

We evaluate Auspex's mutation strategy by mutating various levels of test case, containing chain configurations (L1), block fields (L2), transaction contexts (L3), and contract operations (L4). Table 3 displays that each level contributes uniquely to enhancing code coverage and identifying bugs. Note that when the mutation of L2-4 is turned off, on-chain data cannot be generated and we ignore such two scenarios. Mutating only L2 results in the coverage of 3,249 branches and 1 bug (i.e., bug#6), while integrating all four levels achieves coverage of 14,130 branches and 13 bugs, highlighting the incremental contribution of each level in uncovering more code paths and bugs. Besides, by applying only a single level of chain parameters, Auspex can identify between one and four bugs. Thus, the results in Table 3 confirm Auspex's design choices, emphasizing the importance of holistic mutation strategies in advancing fuzzing coverage and efficiency.

In Table 4, we evaluate the feedback mechanism by a modified version of Auspex, Auspex-nf, which disables the feedback mechanism. Auspex achieves a code coverage of 14,130 branches, a 26.1% improvement over the 10,991 branches covered by Auspex-nf. Moreover, Auspex detects all 13 bugs,

Table 4: The effect of charging-guide fuzzing

Items	Code Coverage	Unidentified Bugs
Auspex-nf	10,991	bug#1,7,13
Auspex	14,130	None

Auspex-nf: Auspex disables feedback mechanism

Table 5: The effect of oracle's differential detection

Items	# of bugs	Unidentified Bugs
Auspex-nd	11	bug#1,bug#9
Auspex	13	None

Auspex-nd: Auspex without differential detection

whereas Auspex-nf fails to uncover bug#1,7,13. These results emphasize the significance of Auspex's feedback mechanism in guiding test case generation, path exploration, and detection capabilities.

We evaluate the differential detection mechanism by comparing Auspex with Auspex-nd, a version of the Auspex without differential detection. As shown in Table 5, Auspex identifies all 13 bugs, while Auspex-nd detects 11 bugs, failing to uncover bug#1,9. This experiment demonstrates that differential detection, by comparing charging behaviors across varying blockchain configurations, is crucial for identifying subtle inconsistencies missed by single-objection approaches, enhancing Auspex's bug detection capabilities.

Answer to RQ3: *The mutation strategy is critical for uncovering complex bugs by generating diverse and high-quality test cases. The feedback mechanism is essential for guiding the exploration of critical execution paths. Differential detection plays a pivotal role in identifying subtle bugs.*

7 Security impact of TFM inconsistency bugs

We further assess the impact of TFM inconsistency bugs on their economic loss (§7.1) and security impact (§7.2).

7.1 Financial loss of the bugs

To understand the security impact of TFM inconsistency bugs, we will measure the financial loss of users caused by five bugs containing bug#1, bug#3-5, and bug#9. We select them as they can cause financial loss to users. Then, we will measure financial loss on Ethereum and BSC. The evaluation focuses on the 1.5M blocks spanning from Ethereum block height #18.5M (Nov-04-2023) to #20M (Jun-01-2024) and BSC block height #38.5M (May-07-2024) to #40M (Jun-28-2024).

First, Table 6 provides the loss measurement rules for these

Table 6: Loss rules of the bugs.

Types	Loss rules
Bug#1	$\frac{^1(tx_{to}=addr_{precompile})\wedge{}^2(len(tx_{inputdata})=0))}{^3g_{loss}\leftarrow\{g_{tx},g_{precompile}\}}$
Bug#3	$\frac{^1tx_{to}=tx_{sender}}{^2g_{loss}\leftarrow\{g_{callvalue}\}}$
Bug#4	$\frac{^1Isrevert(tx)=true}{^2g_{loss}\leftarrow(g_{bug4}-g_{warmaccess})}$ $g_{bug4}\in\{g_{callvalue},g_{newaccount},g_{coldsload},g_{tx},g_{codedeposit}\}$
Bug#5	$\frac{^1tx.writetimes(CA)\geq 2}{^2g_{loss}\leftarrow\{g_{callvalue}\}*0.5-g_{warmaccess}}$
Bug#9	$\frac{^1SAS.read(account)\vee{}^2SAS.read(slot)}{^3g_{loss}\leftarrow(g_{bug9}-F_{SAS}*g_{bug9})}$ $g_{bug9}\in\{g_{acladdress},g_{coldsload},g_{aclstorage},g_{coldaccountaccess}\}$

The above rules also apply in internal transactions.

Table 7: The four Configurations of different nodes.

Configurations	Models [17]	CPU (Cores)	Mem (GB)	Storage (TB)
Datacenter	c5.9xlarge	32	72	2
Devnet	c5.4xlarge	16	32	2
Community	c5.2xlarge	8	16	2
Testnet	c5.xlarge	4	8	2

bugs. By following EIP-2929 [15], we define the user's financial loss as the difference between the gas fee charged to the user and the gas fee corresponding to the actual resources utilized. Then, we interpret the five rules as follows.

(i) Loss of bug#1. Bug#1 occurs when a transaction invokes a precompiled contract with zero-length inputdata, resulting in a financial loss equivalent to g_{tx} or $g_{precompile}$ for the user. Given that executing a precompiled contract requires user-provided parameters, a zero-length parameter indicates an erroneous invocation by the user. If the user sets only 21,000 gas (i.e., g_{tx}) for the transaction, it will be reverted, resulting in a complete loss of g_{tx}. However, if enough transaction fee is set, the user will incur a further loss, $g_{precompile}$, for mistakenly invoking a precompiled contract.

(ii) Loss of bug#3. The bug#3 arises when the sender and receiver addresses are identical. Due to the absence of an actual transfer, the user was erroneously charged the gas fee of ether transferring, thereby incurring a loss of $g_{callvalue}$.

(iii) Loss of bug#4. Bug#4 is characterized by the transaction revert, where the associated state updates remain in memory but will not persist to disk. Consequently, users should only incur the gas fee for memory write, rather than the fee for disk write. Therefore, the loss of bug#4 is the difference between (a) the gas fee of disk write (i.e., $g_{callvalue}$, $g_{newaccount}$, $g_{coldsload}$, g_{tx}, and $g_{codedeposit}$) and (b) the gas fee of memory write (i.e., $g_{warmaccess}$). Note that when we apply the rule of bug#4 on $tx_{external}$, we also need to deduct from the loss the gas fee that the sender's address updates nonce (also applied for the loss rule of bug#3).

(iv) Loss of bug#5. The bug#5 arises when a transaction performs multiple writes to the same account. The block will load the account from the disk to memory at the first write. Despite the subsequent writes being performed in memory, Ethereum charges the memory writes as the gas fee of disk write. Therefore, the user's loss due to bug#5 is the difference between the gas fee of updating an account in disk (i.e., $g_{callvalue} * 0.5$) and its memory updating (i.e., $g_{warmaccess}$).

(v) Loss of bug#9. Bug#9 is triggered as Ethereum deploys SAS to improve the efficiency of disk reading, but does not adjust the related gas fee. Therefore, we introduce F_{SAS} to represent the optimization effect of SAS. The loss of bug#9 is the difference between MPT's gas fee of disk read (i.e., g_{bug9}) and SAS's gas fee of disk read (i.e., $g_{bug9} * F_{SAS}$).

However, determining the value of F_{SAS} is a tough task. Due to the decentralization of blockchains, each node of the blockchain is heterogeneous and has varying hardware configurations, leading to various optimization effects of SAS on different nodes. To address this task, we obtain the optimization effect of F_{SAS} by testing the performance SAS across diverse node configurations. Concretely, we draw inspiration from previous works [43, 57] to set up four node configurations, including datacenter, testnet, devnet, and community. By conducting experiments on AWS nodes [17] with the four configurations, we can derive a reasonable value of F_{SAS}. Table 7 illustrates the four configurations. We conduct experiments in the 1.5M blocks on Ethereum and BSC to test the different performances of blockchain reading data on SAS and MPT, respectively. We observed disk reads of 2.93×10^8 on Ethereum (resp. 8.70×10^8 on BSC) and then we averaged the data obtained from the four configurations. The results show that the disk read latency is optimized from 727 us (resp. 929 us on BSC) to 235 us (resp. 317 us on BSC) by replacing MPT with SAS. Hence, we determine the value of F_{SAS} as 0.32 (=235/727) on Ethereum and 0.34 (=317/929) on BSC.

Table 8: Total impact and loss of the five bugs on 1.5M blocks.

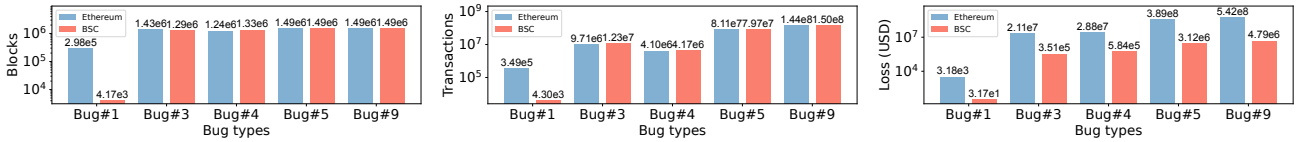
Platforms	Blocks	Txs	Gas	Ether/BNB	Loss fee (USD)
Ethereum	1.49×10^6	1.51×10^8	7.35×10^{12}	3.48×10^5	9.81×10^8
	99.94%	61.50%	32.39%	30.24%	30.43%
BSC	1.49×10^6	1.52×10^8	9.93×10^{12}	1.45×10^4	8.85×10^6
	99.97%	80.42%	43.45%	35.89%	35.91%

We display the impacted amount and its ratio for the total amount.

Then, we adopt the five rules to quantify the financial loss of five bugs. Table 8 shows the blocks, transactions, gas loss, ether/bnb loss, and fee losses affected by the five bugs on 1.5M blocks of Ethereum and BSC . The ratios in Table 8 indicate the proportion of the data in the total. On Ethereum, the bugs affect 1.49M blocks (99.94%), 0.15 billion transactions (61.50%), leading to 7.35 trillion gas loss (32.39%), 0.34M Ether loss (30.24%), and 981M USD fee loss (30.43%). For BSC, the bugs affect 1.49M blocks (99.97%), 0.15 billion transactions (80.42%), and cause 9.93 trillion gas loss (43.45%), 14,563 BNB loss (35.89%), and 8.85M USD fee loss (35.91%).

Furthermore, Fig. 7 visualizes the impact of each bug on Ethereum and BSC, and the amount on the bar indicates the specific affected data. In terms of the affected blocks in Fig. 7a, bug#1 affects far fewer blocks than the other four bugs due to the relatively infrequent usage of precompiled contracts. Moving on to the number of transactions affected in Fig. 7b, for bugs other than bug#1, the number of affected transactions on Ethereum and BSC are close. This similarity suggests that the two platforms might share some common characteristics, e.g., similar contract invocations. Finally, examining the financial losses incurred by users in Fig. 7c, we observe significant discrepancies between Ethereum and BSC. This is due to the difference in gas and cryptocurrency prices between the two platforms.

In summary, the five bugs (i.e., bug#1, bug#3-#5, and bug#9) cause financial losses of 981M USD in Ethereum and



(a) The impacted blocks for each bug (b) The impact transactions for each bug (c) The financial loss for each bug

Figure 7: The impact of each bug on the number of blocks and transactions, as well as the financial losses caused to users.

Algorithm 2: DoS attack algorithm

Input: *BS*: the blockchain system;
Txnums: Number of transactions to generate;
maxDataSize: Maximum length of inputdata;
Output: *Metrics*: The attack impacts on blockchain

1 $txList \leftarrow \{\}$
2 **for** $i = 1$ *to* Txnums **do**
3 $inputData \leftarrow$ `GenerateMaxData`(*maxDataSize*)
4 $tx_i \leftarrow$ `CreateTransaction`(*inputData*)
5 $newAddress \leftarrow$ `GenerateNewAddress`()
6 tx_i.`SetToAddress`(*newAddress*)
7 $txList \leftarrow txList \cup \{tx_i\}$
8 $Metrics \leftarrow BS$.`sendTx`(*txList*)

8.85M USD in BSC. The findings indicate that the bugs have a significant impact on the economic stability of blockchains.

7.2 Security risk of the bugs

To understand the security risk of TFM inconsistency bugs, we will exploit bug#2,13 to launch DoS attack on the Ethereum testnet [25]. We also adopt the same hardware setup as §6.1.

The basic idea of our DoS attack is to flood $tx_{external}$ with inputdata of maximum length to the blockchain. Concretely, bug#2 indicates that the gas fees of inputdata on disk writes (i.e., $g_{txdatanonzero}$ and $g_{txdatazero}$) are relatively low, while bug#13 represents that $tx_{external}$ does not consider the gas fee of writing new addresses into disk (i.e., $g_{newaccount}$). Therefore, as presented in Algorithm 2, we first generate a large number of transactions with inputdata of maximum length (Line 3-4), then set the receiver address of each transaction as the new address (Line 5-6), and finally flood the network with these transactions (Line 7-8). The maximum length of the inputdata is 130,859 bytes, which is the difference between (i) the maximum length of a transaction (i.e., 128 KB) and (ii) the 213 bytes required fields of the transaction (i.e., nonce, gas tip, gas limit, recipient, value, and signature) [28].

Furthermore, there exists a tradeoff in generating the inputdata between attack cost and impact. Concretely, the most cost-effective method to construct inputdata is to use all-zero bytes, as zero bytes only incur 4 gas (i.e., $g_{txdatazero}$), whereas non-zero bytes incur 16 gas (i.e., $g_{txdatanonzero}$). However, Ethereum employs the algorithm of snappy compression [31] for data storage [77] and transmission [23], which can mitigate the attack impact of all-zero inputdata.

To find a better approach to construct the inputdata, we conduct the following experiments. First, we utilize the cryptographic interface [30] to generate hash values as inputdata because the hash values with high entropy [55] are typically hard to compress [67]. Then, we progressively replace the data in inputdata with zero bytes in a randomized position to reduce gas cost, and compress the hash values with the snappy algorithm. Meanwhile, we collect the compression ratio of the inputdata after being compressed and the average gas cost per byte of the inputdata. Fig. 8 displays the relationship between the compression ratio (blue line) and the gas cost per byte (orange line). When the replacement has not started, the compression ratio is close to 1 (i.e., hard to compress), and the gas cost per byte is close to 16 gas (i.e., $g_{txdatanonzero}$). After the replacement finishes, the compression ratio is close to 0 (i.e., easy to compress), and the gas fee per byte tends to be 4 gas (i.e., $g_{txdatazero}$). Hence, we select the maximum difference between the compression ratio and gas cost per byte to achieve the balance in attack impact and cost (red arrows). Concretely, we choose to replace 67,173 zero bytes into the inputdata, which achieves the compress ratio of 0.90 and gas fee of 9.82 per bytes.

We launch DoS attack ranging from block height #1,888,063 to #1,888,068 on Holesky testnet. To limit the impact of the attack and ensure it does not disrupt the operation of the testnet, the attack scope was confined to six blocks. We select block size and wait time of the transaction as the metrics to evaluate the attack impact. Fig. 9 provides the specific results, which will be illustrated as follows.

(i) Block size. DoS attack constructs transactions with excessively long inputdata that significantly increase the block size, leading to greater network bandwidth and storage resource utilization. Fig. 9a displays trends of block sizes during the attack. We established the baseline by constructing chains through the selection of blocks before the attack. The

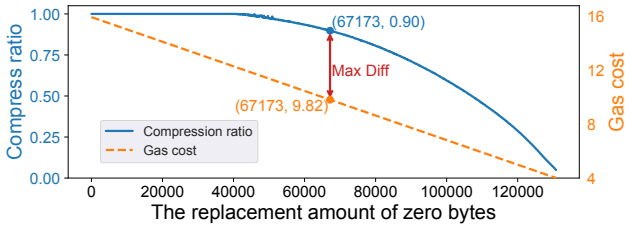
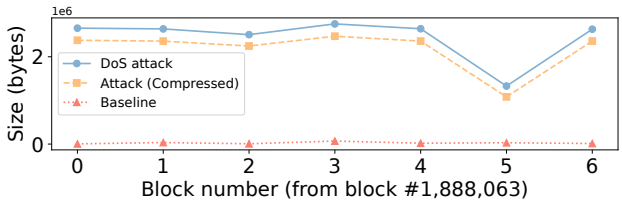
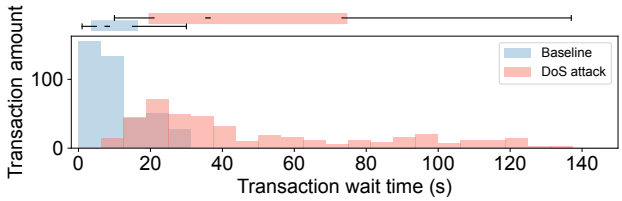


Figure 8: Randomly replace the data at a certain position of inputdata with zero bytes, and observe the changes in the compression rate of inputdata and the gas cost of each byte of inputdata by the snap algorithm.



(a) Impacted block size



(b) Impacted transactions wait time

Figure 9: The impact of DoS attack on block size and transaction wait time. We launch DoS attack in six blocks on testnet (#1,888,063 to #1,888,069). DoS attack amplifies block size 96 times and prolongs the wait time of transactions 4.5 times.

blue line represents the block size under attack, showing a significant increase 96 times on average compared to the baseline (red line), where normal blocks without the attack are processed. Besides, comparing compressed (orange line) and uncompressed block sizes (blue line), DoS attack achieves a compressed ratio of 0.88 on average. Overall, the observations show that DoS attack can amplify the block size even in the affection of a compressed algorithm, consuming the resources of storage and network.

(ii) Wait time of transaction. DoS attack will broadcast a large number of transactions, leading to an increased workload on the blockchain, which in turn leads to prolonged confirmation time (i.e., wait time) for all transactions. Hence, we measure the wait time of transactions during the attack. Then we interpret the method to get the wait time. Concretely, we first deploy three blockchain nodes in Europe, Asia, and America to collect the T_{onpool} indicating the times of transactions entering the pool. We will take the average of T_{onpool} from the three nodes. Besides, we take the timestamp in the block header as the transaction on-chain time (i.e., $T_{onchain}$). Finally, we obtain the wait time of $T_{waittime}$ from the difference between $T_{onchain}$ and T_{onpool} (i.e., $T_{waittime} = T_{onchain} - T_{onpool}$). Fig. 9b presents the distribution of waiting times from the number of transactions during the attack. We constructed the baseline from the block height before DoS attack, using the same number of blocks as in our attack range. Fig. 9b shows a clear distinction in transaction wait times between the baseline (blue bins) and DoS attack scenarios (red bins). The average wait time expands during the attack from 10s to 50s with a marked increase in the number of delayed transactions. Under DoS attack, the distribution of transaction wait times shifts significantly, with the median increasing from 8s in the baseline to 36s during the attack. This indicates that DoS attack effectively introduces 4.5 times =(36/8) delay of transactions, degrading overall blockchain performance.

Table 9: The cost of DoS attack per block.

Platform	Ethereum	BSC	Avalanche	Polygon	Optimism
Gas limit	30M	120M	15M	30M	30M
Gas price (GWei)	26	4.8	39	131	0.15
Coin price (USD)	3,140	493	38	0.82	3.03
Attack cost (USD)	2,480	287	22	3.23	0.014

We adopt the average gas price and coin price during H1 2024.

Economic feasibility. We evaluate the economic feasibility of a DoS attack on five popular blockchain platforms with top market capitalization, including Ethereum, BSC, Avalanche, Polygon, and Optimism. The financial cost of DoS attack is considered that the attacker fully utilizes all gas in the block, reflecting realistic attack scenarios where adversaries aim to fill the block with the transaction with maximum length. Table 9 presents the attack cost in five popular blockchains, detailing the costs of gas limit, gas price, and coin price. We adopt the average gas price and coin price during the first half of 2024. The cost of DoS attack shows significant variance across different blockchain platforms. For example, the cost is highest on Ethereum (2,480 USD per block) due to its high gas price and coin price, while the attack cost is minimal (0.014 USD per block) on Optimism, largely due to its low gas price and relatively lower coin price. The analysis reveals that while high-cost platforms like Ethereum offer greater security through economic deterrence, lower-cost platforms like Optimism are at greater risk of being targeted due to the minimal financial burden on attackers. Unfortunately, most blockchain platforms do not have the high coin prices as Ethereum and BSC. This raises concerns about the resilience of these networks under adversarial conditions, especially as Layer 2 solutions like Optimism grow in adoption [51].

To further investigate the cost and impact of our DoS attack, we compare it with the suppression attack [68]. The suppression attack is also a genus of DoS attack, which has been identified to delay transactions to obtain huge profits in MEV (Maximal Extractable Value) scenarios in the real world [63, 68, 79]. The fundamental design of the suppression attack is similar to our DoS attack in terms of exhausting a block's gas limit. The primary difference between the two attacks is that the suppression attack specifically exploits the opcode `SSTORE` to consume gas and incur an I/O burden. To conduct this comparison, we set up two private chain environments using identical configurations to the mainnet. Each environment consists of 100 blocks, with each block's gas limit set to 30M gas. The experiments are designed to evaluate both attacks under controlled conditions. For the suppression attack, transactions are constructed to perform repetitive `SSTORE` operations targeting different storage slots to maximize gas consumption. In contrast, our DoS attack utilizes a custom calldata into the blockchain.

Table 10: Comparison of attack cost and impact

Items	Gas cast	Payload length	compression ratio
Our DoS attack	2.97M	2.99×10^6B	0.88
Suppression [68]	2.98M	42,351B	0.65

Table 10 displays the results. The costs of both attacks are nearly identical, aligning with their shared design of depleting the gas limit of a block (i.e., 30M gas). However, our DoS attack demonstrates superior effectiveness by achieving higher payload throughput and exerting a greater impact on block resources. Specifically, our attack is able to write approximately 2.99×10^6 bytes of data per block on average, compared to only 42,351 bytes in the suppression attack (i.e., 69× improvement). Additionally, the payload design of our DoS attack keeps a higher proportion of the original data size with a compression rate of 0.88, as opposed to the suppression attack which compresses more aggressively at a rate of 0.65.

There are two reasons for the superior effectiveness of our DoS attack: (i) Our attack achieves higher payload efficiency, allowing significantly more data to be written within the same gas cost. Concretely, the gas cost of our DoS attack per byte written is only 9.82, significantly lower than the suppression attack's 690 gas per byte (opcode `SSTORE` writes 32 bytes payload with 22,100 gas, comprising 20,000 for g_{sset} and 2,100 for $g_{coldsload}$). (ii) The payload design of our DoS attack is compression-resistant, whereas the payloads generated in the suppression attack often contain sequences of zero bytes used for padding. These zero-byte sequences are susceptible to compression, reducing their impact and the overall effectiveness of the suppression attack. The evaluation underscores not only the superior efficiency of our DoS attack but also the critical role that TFM inconsistency bugs play in enabling such attacks. These bugs allow attackers to exploit discrepancies in gas costing and resource utilization, highlighting the critical consequences of TFM bugs being exploited.

In summary, our DoS attack exploiting bug#2 and bug#13 amplified the block size by 96 times and prolonged the waiting time of transactions by 4.5 times, highlighting the security risk of TFM inconsistency bugs for blockchains.

8 Discussion

In this section, we will discuss some considerations of our work.

Security implications. Beyond the impact on financial losses (§7.1) and security (§7.2), TFM inconsistency bugs also pose significant risks to the security of the Ethereum ecosystem in the following three aspects: (i) Reputational damage and user attrition. The presence of fee inconsistency bugs erodes user confidence in Ethereum's reliability as a decentralized platform. Such erosion of trust could result in reduced adoption rates and weaken the network effect that underpins the growth and resilience of the ecosystem. (ii) Economic barriers and reduced accessibility. The inefficiencies introduced by fee inconsistencies impose undue financial burdens on users, particularly those with limited resources. Elevated or unpredictable transaction costs create a barrier to entry for fund-constrained participants, thereby limiting the inclusivity of the network. (iii) Impact on developer incentives. TFM inconsistencies further deter developers from maintaining or building new DApps. Uncertain gas fees make it difficult for developers to estimate the transaction costs required for executing new features, reducing their willingness to invest in creating innovative applications.

Inherent mechanism and bugs TFM inconsistency bugs arise from the core design of the blockchain and are not influenced by the underlying operating system or hardware. This is because TFM and related data structures (e.g., accounts, slots, MPT, and precompiled contracts) are fully contained within the blockchain design and specification itself. Blockchain systems follow deterministic, platform-independent rules set by their consensus protocols, and it indicates that TFM inconsistency bugs are inherent in blockchain nodes. Therefore, these TFM inconsistency bugs are essentially blockchain protocol-level issues inherent to the design of the blockchain.

Client diversity. While Auspex identified 13 TFM inconsistency bugs within the geth, the broader Ethereum ecosystem relies on multiple client implementations (i.e., client diversity) [64, 65], which include Nethermind [29], Besu [18], and Erigon [24], each developed using different languages and architectures. Given the critical role of client diversity in ensuring network resilience, it is crucial to assess whether these bugs persist across different implementations. Upon extending our analysis to these additional Ethereum clients, we confirmed that all of them are indeed susceptible to the TFM inconsistency bugs, demonstrating that the issue is inherent to the consensus logic rather than specific to get. This finding underscores that the bugs are not isolated to a single codebase but are a systemic issue affecting the Ethereum ecosystem as a whole, irrespective of client diversity.

Bug fix. Addressing TFM inconsistency bugs requires protocol upgrades through a hard fork. Concretely, the first step involves identifying TFM inconsistency bugs based on our work. Next, several Ethereum improvement proposals (e.g., EIP-2929) are drafted to fix these inconsistencies. Once those proposals are reviewed and approved by the community, developers implement the updates in the blockchain clients. Finally, the network executes the hard fork at a predetermined block height, eliminating inconsistencies and thereby enhancing fairness and security across the ecosystem.

Loss optimization. The financial losses defined in §7.1 will eventually be paid to miners/validators through transaction fees. If Ethereum optimizes corresponding losses, miners/validators' revenue will be reduced, but it will not compromise Ethereum's economic incentive mechanism. According to the experience of EIP-1559 [52] and EIP-2929 [15], miners/validators have shown a willingness to accept a decrease in their revenue as long as it benefits the overall ecosystem [52].

Byte-array mutation. Auspex utilizes a byte-array mutation strategy for elements within test cases (e.g., inputdata) without considering the structural syntax of these elements. While this approach is efficient, it may not be optimal in some scenarios. For instance, Confuzz [71] introduces a mutation strategy that leverages the specific types of contract arguments for inputdata. Although such type-aware mutation strategies can enhance precision, they are non-trivial in implementing and introducing additional overhead (e.g., recovering the types from bytecode). Our evaluation results also demonstrate that our byte-array mutation is effective in finding bugs [75].

The lessons of bugs. The bugs found across the Ethereum expose several key lessons that highlight broader challenges within blockchain systems. (i) Deviations from specifications can result in incorrect behavior, as seen in cases like bug#3 and bug#4, where gas fees are charged despite the ether transfer not happening. (ii) Mismatches between resource usage and pricing can lead to unfair charges, exemplified by bug#5, where memory accesses are incorrectly charged as disk accesses. (iii) Design oversights, such as deploying precompiled contracts at addresses commonly mistaken for burn addresses (i.e., `0x1` to `0xA`), show that poor design choices can exacerbate the financial loss of users. (iv) Outdated gas pricing that fails to account for optimizations (like the introduction of SAS in bug#9) underscores the risks of not regularly adjusting cost structures in response to evolving system efficiencies.

9 Related work

TFM. Several studies [41, 52, 60–62, 76] have focused on TFM of blockchain. Chen et al. [37] measure the resource cost of TFM of Ethereum, and propose an adaptive TFM to deter the potential DoS attack by dynamically adjusting the gas fee of the opcodes. Brokenmetre [61] constructs the benchmark to further find under-price opcode in Ethereum. Furthermore, Brokenmetre demonstrated the designed contracts with opcodes that execute at a significantly slower speed can be used to launch DoS attacks, thereby impairing the real-time capabilities of the blockchain. Liu et al. [52] conduct the empirical analysis on EIP-1559, which is a hard-fork upgrade for TFM on Ethereum. They found that the deployment of EIP-1559 for TFM could optimize the gas price and wait time of transactions.

Differentiating from prior work that relies on manual benchmark design and data analysis, Auspex first automates the finding of inconsistency bugs in TFM, thereby enabling comprehensive explorations of a broader scenarios of TFM.

Fuzzing. Fuzzing, as a popular test technique, has been applied in many works [39, 40, 56, 70, 75] to detect bugs in blockchain systems [73]. Fluffy [75] employs differential fuzzing technology to detect consensus bugs in Ethereum. It mutates and executes multiple transactions across different Ethereum implementations, identifying bugs through cross-comparison of the resulting blockchain states. Tyr [40] focuses on detecting consensus failure bugs in six blockchains, including Fabric, FISCO-BCOS, Quorum, Diem, Ethereum, and EOS. Tyr will first simulate an entire blockchain network with several nodes and then broadcast the mutated network messages related to blockchain consensus in the network. Tyr finds consensus failure bugs by comparing the inconsistency (e.g., block height and blockchain state) between different nodes in their networks. MPFuzz [70] concentrates on uncovering the bugs exploited by DoS attacks in Ethereum txpool. The developers of MPFuzz first construct some semantics patterns of Ethereum transactions in txpool, and then utilize the pattern to generate test cases to test txpool to trigger DoS attacks.

However, current works [39, 40, 56, 70, 75] are limited in their ability to detect TFM inconsistency bugs effectively. (i) They do not concentrate on changes in gas fees and resource consumption, whereas Auspex leverages these data as feedback and oracle. (ii) TFM inconsistency bugs do not typically cause system crashes or consensus-level inconsistencies, but Auspex is equipped with a corresponding oracle to identify such issues. (iii) They neglect chain configurations and may omit to explore some blockchain components. In contrast, Auspex can explore more code logic through mutating chain configurations.

10 Conclusion

This paper presented Auspex, the first automated fuzzing tool designed to detect TFM inconsistency bugs in Ethereum. Auspex introduces a chain-based test case generation strategy, a charging-guided fuzzing approach, and two general oracles (FCP and RCP) to identify TFM inconsistency bugs. Our evaluation uncovered 13 new bugs and demonstrated 3.5 times more code branch coverage than state-of-the-art tool. We further analyzed the financial and security impact of these bugs. It reveals these bugs cause exceeding multi-million-dollar losses on Ethereum and BSC. Besides the DoS attack exploiting these bugs increases block size by 96 times and the transition wait time by 4.5 times. These results emphasize the critical need for automated solutions like Auspex to detect TFM inconsistency bugs.

Acknowledgements. We thank anonymous reviewers for their comments and our shepherd. We would also like to thank Julian Ma and Ansgar Dietrichs from the Ethereum Foundation; Their advice is of great help to our work. Besides, the support of Martin Monperrus and Javier Ron Arteaga from KTH Royal Institute of Technology is also important for us. This work was partly supported by National Natural Science Foundation of China (U2336204, 62332004), Hong Kong RGC Projects (PolyU15222320, PolyU15231223, and Theme-based Research Scheme Project T43-513/23-N), National Key R&D Program of China (2022YFB3103500), Sichuan Science and Technology Program (2024NSFTD0031, 2024YFHZ0339), Sichuan Provincial Natural Science Foun-

dation for Distinguished Young Scholars (2023NSFSC1963), and Fundamental Research Funds for Chinese Central Universities (ZYGX2021J018).

Ethics considerations

In this study, we carefully addressed ethical concerns associated with identifying and reporting TFM inconsistency bugs, following established best practices for responsible disclosure. Our efforts are detailed below: **(i)** After completing the manuscript, we promptly reported the identified bugs to the official Ethereum development teams. They have acknowledged both the bugs, and our methodology and tools, praising our findings as significant issues that were overlooked by prior research. Given the complexity of the identified bugs, we are actively collaborating with developers to explore effective mitigation strategies. **(ii)** To prevent potential exploitation, we exclusively disclosed the bug details to Ethereum's core developer teams. The technical specifics of the bugs were not made publicly available to ensure adversaries could not exploit these vulnerabilities before mitigation efforts were in place. **(iii)** The performance evaluation of Auspex and the assessment of bug exploitation impacts were conducted strictly within controlled environments, including our local setup and the Ethereum Holesky testnet. At no point did our experiments involve the Ethereum mainnet, ensuring that the research did not disrupt real-world entities.

Open science

We embrace the principles of open science and have made the data and source code of our work publicly available at https://zenodo.org/records/14712667. Our repository contains comprehensive documentation for both configuration details and execution procedures.

References

[1] The abnormal transaction of precompiled contract. https://etherscan.io/tx/0x1fb0c35040c9ddf3929a3f3ad40f6fdb9cd2996475e9d3d216da94e9f6805d05.

[2] The abnormal transaction of precompiled contract. https://etherscan.io/tx/0x6b01625d57a71c5c3e55e4b5757ba1896a4e5548ba6556cfbf674a9c5d16960d.

[3] The abnormal transaction of precompiled contract. https://etherscan.io/tx/0xe5037c45fad6726fbeae3329c2edf1e6274238ee2befee7e4c221e837b26df2c.

[4] Avalanche Chain. https://www.avax.network/.

[5] BNB Smart Chain (BSC). https://www.bnbchain.org/.

[6] Bonsai tree. https://besu.hyperledger.org/public-networks/concepts/data-storage-formats/.

[7] The Ethereum ecosystem. https://chainlist.org/.

[8] Etherscan: the blockchain explorer. https://etherscan.io/.

[9] Flat layout database. https://github.com/NethermindEth/nethermind/tree/feature/state_db_layout_by_path.

[10] The introduction of precompiled contract. https://www.evm.codes/precompiled.

[11] Optimism. https://www.optimism.io/.

[12] Plain state. https://github.com/ledgerwatch/erigon/blob/devel/docs/programmers_guide/db_walkthrough.MD#table-plainstate.

[13] Polygon Chian. https://polygon.technology/.

[14] Shanghai dos attack. https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack.

[15] Eip-2929: Gas cost increases for state access opcodes. https://eips.ethereum.org/EIPS/eip-2929, 2021.

[16] Eip-2930: Optional access lists. https://eips.ethereum.org/EIPS/eip-2930, 2021.

[17] Amazon ec2 c5 instances. https://aws.amazon.com/ec2/instance-types/c5/?nc1=h_ls, 2024.

[18] Besu, java implementation of ethereum. https://github.com/hyperledger/besu, 2024.

[19] The deployment of snapshot acceleration structure. https://blog.ethereum.org/2021/03/03/geth-v1-10-0, 2024.

[20] The document of solidity language. https://docs.soliditylang.org, 2024.

[21] Eip-2200: Structured definitions for net gas metering. https://eips.ethereum.org/EIPS/eip-2200, 2024.

[22] Eip-4895: Beacon chain push withdrawals as operations. https://eips.ethereum.org/EIPS/eip-4895, 2024.

[23] Eip 706: Devp2p snappy compression. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-706.md, 2024.

[24] Erigon, golang implementation of ethereum. https://github.com/ledgerwatch/erigon, 2024.

[25] Ethereum holešky testnet. https://github.com/eth-clients/holesky, 2024.

[26] go-fuzz: randomized testing for go. https://github.com/dvyukov/go-fuzz, 2024.

[27] The implementation of etc. https://github.com/etclabscore/core-geth, 2024.

[28] The maximum length of ethereum transaction's inputdata. https://github.com/ethereum/go-ethereum/blob/6eb42a6b4f9a07a4a11a9ac706b6a738212de210/core/txpool/legacypool/legacypool_test.go#L1258, 2024.

[29] Nethermind, c sharp implementation of ethereum. https://www.nethermind.io/, 2024.

[30] secrets — generate secure random numbers for managing secrets. https://docs.python.org/3/library/secrets.html, 2024.

[31] Snappy compression. https://en.wikipedia.org/wiki/Snappy_(compression), 2024.

[32] Snapshot acceleration structure. https://blog.ethereum.org/2020/07/17/ask-about-geth-snapshot-acceleration, 2024.

[33] Amjad Aldweesh, Maher Alharby, Maryam Mehrnezhad, and Aad van Moorsel. The opbench ethereum opcode benchmark framework: Design, implementation, validation and experiments. *Performance Evaluation*, 2021.

[34] Sofia Bobadilla, Monica Jin, and Martin Monperrus. Do automated fixes truly mitigate smart contract exploits? *arXiv preprint arXiv:2501.04600*, 2025.

[35] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *SP*, 2022.

[36] Vitalik Buterin. Some medium-term dust cleanup ideas. https://ethereum-magicians.org/t/some-medium-term-dust-cleanup-ideas/6287, 2024.

[37] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *ISPEC 2017*, 2017.

[38] Ting Chen, Yuxiao Zhu, Zihao Li, Jiachi Chen, Xiaoqi Li, Xiapu Luo, Xiaodong Lin, and Xiaosong Zhange. Understanding ethereum via graph analysis. In *INFOCOM*, 2018.

[39] Weimin Chen, Xiapu Luo, Haipeng Cai, and Haoyu Wang. Towards smart contract fuzzing on gpu. In *SP*, 2024.

[40] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Yu Jiang, Ting Chen, and Jiaguang Sun. Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model. In *SP*, 2023.

[41] Hao Chung and Elaine Shi. Foundations of transaction fee mechanism design. In *SODA*, 2023.

[42] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. etainter: detecting gas-related vulnerabilities in smart contracts. In *ISSTA*, 2022.

[43] Vincent Gramoli, Rachid Guerraoui, Andrei Lebedev, Chris Natoli, and Gauthier Voron. Diablo: A benchmark suite for blockchains. In *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023.

[44] Zheyuan He, Zihao Li, Ao Qiao, Xiapu Luo, Xiaosong Zhang, Ting Chen, Shuwei Song, Dijun Liu, and Weina Niu. Nurgle: Exacerbating resource consumption in blockchain state storage via mpt manipulation. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[45] Zheyuan He, Shuwei Song, Yang Bai, Xiapu Luo, Ting Chen, Wensheng Zhang, Peng He, Hongwei Li, Xiaodong Lin, and Xiaosong Zhang. Tokenaware: Accurate and efficient bookkeeping recognition for token smart contracts. *ACM Transactions on Software Engineering and Methodology*, 2023.

[46] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *CSF*, 2018.

[47] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *SP*, 2020.

[48] Hanna Kim, Jian Cui, Eugene Jang, Chanhee Lee, Yongjae Lee, Jin-Woo Chung, and Seungwon Shin. Drainclog: Detecting rogue accounts with illegally-obtained nfts using classifiers learned on graphs. In *NDSS*, 2024.

[49] Shinhae Kim and Sungjae Hwang. Etherdiffer: Differential testing on rpc services of ethereum nodes. In *Proceedings of the 31st ACM Joint European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering, 2023.

[50] Yeonsoo Kim, Seongho Jeong, Kamil Jezek, Bernd Burgstaller, and Bernhard Scholz. An {Off-The-Chain} execution environment for scalable testing and profiling of smart contracts. In *USENIX ATC*, 2021.

[51] Zihao Li, Xinghao Peng, Zheyuan He, Xiapu Luo, and Ting Chen. famulet: Finding finalization failure bugs in polygon zkrollup. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 971–985, 2024.

[52] Yulin Liu, Yuxuan Lu, Kartik Nayak, Fan Zhang, Luyao Zhang, and Yinhong Zhao. Empirical analysis of eip-1559: Transaction fees, waiting times, and consensus security. In *CCS*, 2022.

[53] Feng Luo, Huangkun Lin, Zihao Li, Xiapu Luo, Ruijie Luo, Zheyuan He, Shuwei Song, Ting Chen, and Wenxuan Luo. Towards automatic discovery of denial of service weaknesses in blockchain resource models. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[54] Feng Luo, Ruijie Luo, Ting Chen, Ao Qiao, Zheyuan He, Shuwei Song, Yu Jiang, and Sixing Li. Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[55] Shoufu Luo, Jeremy D Seideman, and Sven Dietrich. Fingerprinting cryptographic protocols with key exchange using an entropy measure. In *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2018.

[56] Fuchen Ma, Yuanliang Chen, Meng Ren, Yuanhang Zhou, Yu Jiang, Ting Chen, Huizhong Li, and Jiaguang Sun. Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols. In *NDSS*, 2023.

[57] Liyuan Ma, Xiulong Liu, Yuhan Li, Chenyu Zhang, Gaowei Shi, and Keqiu Li. Gfbe: A generalized and fine-grained blockchain evaluation framework. *IEEE Transactions on Computers*, 2024.

[58] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 1998.

[59] Satoshi Nakamoto and A Bitcoin. A peer-to-peer electronic cash system. *Bitcoin.–URL: https://bitcoin.org/bitcoin. pdf*, 2008.

[60] Michael Pacheco, Gustavo A Oliva, Gopi Krishnan Rajbahadur, and Ahmed E Hassan. What makes ethereum blockchain transactions be processed fast or slow? an empirical study. *Empirical Software Engineering*, 2023.

[61] Daniel Perez and Benjamin Livshits. Broken metre: Attacking resource metering in evm. *NDSS*, 2019.

[62] Giuseppe Antonio Pierro and Henrique Rocha. The influence factors on ethereum transaction fees. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019.

[63] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *SP*, 2022.

[64] Javier Ron, Zheyuan He, and Martin Monperrus. Proving and rewarding client diversity to strengthen resilience of blockchain networks. *arXiv preprint arXiv:2411.18401*, 2024.

[65] Javier Ron, César Soto-Valero, Long Zhang, Benoit Baudry, and Martin Monperrus. Highly available blockchain nodes with n-version design. *TDSC*, 2023.

[66] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[67] MTCAJ Thomas and A Thomas Joy. *Elements of information theory*. Wiley-Interscience, 2006.

[68] Christof Ferreira Torres, Ramiro Camino, et al. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain. In *USENIX Security 21*, 2021.

[69] Weijie Wang, Yujie Lu, Charalampos Papamanthou, and Fan Zhang. The locality of memory checking. In *CCS*, 2023.

[70] Yibo Wang, Yuzhe Tang, Kai Li, Wanning Ding, and Zhihua Yang. Understanding ethereum mempool security under asymmetric dos by symbolized stateful fuzzing.

[71] Taiyu Wong, Chao Zhang, Yuandong Ni, Mingsen Luo, HeYing Chen, Yufei Yu, Weilin Li, Xiapu Luo, and Haoyu Wang. Confuzz: Towards large scale fuzz testing of smart contracts in ethereum. In *INFOCOM*, 2024.

[72] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[73] Shuohan Wu, Zihao Li, Luyi Yan, Weimin Chen, Muhui Jiang, Chenxu Wang, Xiapu Luo, and Hao Zhou. Are we there yet? unraveling the state-of-the-art smart contract fuzzers. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[74] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing ethereum's gas mechanism. In *EuroS&PW*, 2019.

[75] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *OSDI*, 2021.

[76] Abdullah A Zarir, Gustavo A Oliva, Zhen M Jiang, and Ahmed E Hassan. Developing cost-effective blockchain-powered applications: A case study of the gas usage of smart contract transactions in the ethereum blockchain platform. *TOSEM*, 2021.

[77] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. Compressdb: Enabling efficient compressed data direct processing for various databases. In *Proceedings of the 2022 International Conference on Management of Data*, 2022.

[78] Kunsong Zhao, Zihao Li, Jianfeng Li, He Ye, Xiapu Luo, and Ting Chen. Deepinfer: Deep type inference from smart contract bytecode. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 745–757, 2023.

[79] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. High-frequency trading on decentralized on-chain exchanges. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

A Gas fees and its related operations

We list the gas fees and related operations in Table 11. Note that we only provide the opcodes related to our work.

Table 11: The notions of gas fees and related operations.

Name	Related operations
$g_{verylow}$	PUSH.
$g_{warmaccess}$	SLOAD, SSTORE, BAlANCE, EXTCODESIZE,EXTCODECOPY, CALL, EXTCODEHASH, DELEGATECALL, STATICCALL and CALLCODE.
$g_{coldaccountaccess}$	BALANCE, EXTCODESIZE, EXTCODECOPY, CALL, EXTCODEHASH, DELEGATECALL STATICCALL and CALLCODE.
$g_{coldsload}$	SLOAD and SSTORE.
g_{sset}	SSTORE .
$g_{acladdress}$	BALANCE, EXTCODESIZE, EXTCODECOPY, EXTCODEHASH, CALL, DELEGATECALL, STATICCALL and CALLCODE.
$g_{aclstorage}$	SLOAD and SSTORE.
$g_{callvalue}$	CALL and CALLCODE.
$g_{newaccount}$	CALL.
g_{tx}	$tx_{external}$.
$g_{txdatazero}$	$tx_{external}$.
$g_{txdatanonzero}$	$tx_{external}$.
$g_{codedeposit}$	CREATE and CREATE2
$g_{precompiled}$	N.A.

B Uncovered bugs

In this section, we will illustrate the other six bugs (i.e., bug#6-9 and bug#10-12) identified by Auspex

Bug#6. Bug#6 is triggered when a validator or miner receives block rewards, which involves updating their account balance on disk without incurring the corresponding gas fee for disk writing. Specifically, when block rewards are allocated, the blockchain modifies the recipient's balance by updating the state and writing the updates directly to the disk. Unlike typical state updates triggered by user transactions, which are charged based on the resources consumed (e.g., memory and disk write), the disk operations related to block rewards are not associated with any gas fee [22]. The Bug#6 holds in FIB as $FIB_{bug6} \models (g_{\varnothing} \rightarrow r_{diskw}^{len(account)}) \wedge (g_{callvalue} \rightarrow r_{diskw}^{len(account)}) \wedge (g_{\varnothing} < g_{callvalue})$.

Bug#7. Bug#7 is triggered by the failure to charge gas fees for disk reads during the execution of the SSTORE code. Although SSTORE is primarily considered a state-writing, the Ethereum specification (i.e., EIP-2200 [21]) introduces a new logic that requires first reading the disk before modifying the value. Specifically, when SSTORE is invoked, it reads the existing value from the disk. If the value to be stored matches the one already present, no disk write is performed, avoiding unnecessary state updates in disk. However, if the values differ, the state is updated, but no gas fee is charged for the first disk read operation. Bug#7 holds in FIB as $FIB_{bug7} \models (g_{\varnothing} \rightarrow r_{diskr}^{len(slot)}) \wedge (g_{coldsload} \rightarrow r_{diskr}^{len(slot)}) \wedge (g_{\varnothing} < g_{coldsload})$.

Bug#8. Bug#8 is due to inconsistent gas fees for memory reads between the SLOAD opcode and other memory opcodes like MLOAD. Typically, SLOAD is considered a disk-read operation that retrieves contract slot data from the blockchain state. However, under the latest Ethereum specification (i.e., EIP-2929 [15]), if the slot has already been accessed once (i.e., it caches in memory), subsequent reads of that slot by SLOAD opcode only incurs gas fee of 100 (i.e., $g_{warmaccess}$) for memory access. In contrast, memory operations like MLOAD only charge 3 gas for reading data from memory. This discrepancy results in an overcharging issue where SLOAD incurs a significantly higher fee for the same memory read operation as MLOAD. Bug#8 holds in FIB as $FIB_{bug8} \models (g_{verylow} \rightarrow r_{memr}^{len(slot)}) \wedge (g_{warmaccess} \rightarrow r_{memr}^{len(slot)}) \wedge (g_{verylow} < g_{warmaccess})$.

Bug#10. Bug#10 is due to when account updates in disk occur without charging a gas fee for those updates. Specifically, Bug#10 arises in scenarios where a user sends $tx_{external}$ to contract A, which in turn makes an internal invoke to contract B. If contract B modifies a slot in its storage, the corresponding storage root in contract B's account must be updated on disk. However, no gas fees are charged for this account B update, leading to an inconsistency. In summary, the bug occurs because the storage trie modification of contract B incurs no additional gas fee for updating its account state. Bug#10 can be classified as a FIB, expressed in $FIB_{bug10} \models (g_{\varnothing} \rightarrow r_{diskw}^{len(account)}) \wedge (g_{callvalue} \rightarrow r_{diskw}^{len(account)}) \wedge (g_{\varnothing} < g_{callvalue})$.

Table 12: The chain configurations

Items	Options
CacheConf	Trie cache size, SAS cache size, and cache live time.
DBConf	Hash db and path db.
ForkConf	All hard forks of Ethereum.

Bug#11. Bug#11 is due to reading bytecode from disk incurs no gas fee. Specifically, when executing `Ext*` opcodes (e.g., `EXTCODECOPY`), the blockchain first accesses the account in the state from disk and then reads the associated bytecode. However, `Ext*` opcodes only charge a gas fee of 2,600 (i.e., $g_{coldaccountaccess}$), similar to the `BALANCE` opcode. It indicates that the gas fees only consider the cost of reading the account information and ignore the resource of retrieving bytecode information from the disk. In summary, although `Ext*` opcodes involve both account and bytecode reads, they only charge for the account read. Bug#11 can be expressed in RIB as $RIB_{bug11} \models (g_{coldaccountaccess} \rightarrow r_{diskr}^{len(account)}) \wedge (g_{coldaccountaccess} \rightarrow (r_{diskr}^{len(account)}, r_{diskr}^{len(code)})) \wedge (r_{diskr}^{len(account)} < (r_{diskr}^{len(account)}, r_{diskr}^{len(code)}))$.

Bug#12. Bug#12 is triggered when gas fees are charged for disk reads that never actually occur. Specifically, this issue arises when users define accounts or storage slots in the access list of a transaction, but those accounts or slots are not accessed during the transaction execution (e.g., transaction `0x0dd0c0`[1]). Despite no disk access, blockchain still charges gas fees like the resources were used. In summary, despite no disk access, gas fee is still charged based on the access list entries. Bug#12 is represented in RIB as $RIB_{bug12} \models (g \rightarrow r_{diskr}^{len(account)}) \wedge (g \rightarrow r_{\varnothing}^{0}) \wedge (r_{\varnothing}^{0} < r_{diskr}^{len(account)})$, where $g \in \{g_{acladdress}, g_{aclstorage}\}$.

C The chain configurations

We list the chain configurations in Table 12.

D Function signatures extraction

This section explains the method for extracting 8-byte smart contract function signatures from bytecode. First, we disassemble the contract and then utilize the function signature selector at the beginning of the bytecode to extract the corresponding function signatures. Specifically, at the start of the contract bytecode, the `PUSH` opcode with `EQ` opcode is used to compare the first eight bytes of inputdata [71, 78]. This comparison controls the jump to different functions, and we extract the function signature from the pattern formed by `PUSH` and `EQ` opcodes.

[1] https://etherscan.io/tx/0x0dd0c0f90b2ffbe5588f44116a9b30a2f978cc562e98e70c37ddb2ba738668e4