Mamori: The Automated Smart Contract Auditing System

Andy M. Lee[*]
Mamori

February 1, 2024[†]

Abstract

Mamori is a Web3 algorithmic smart contract auditing system focusing on zero-day economic exploits and MEV (Maximal Extractable Value), addressing the challenges of scalability, automation, exploit detection relevance, and efficacy in the Web3 security landscape. Our mission is to applly interdisciplinary technologies and protect against the 'unknown unknowns' vulnerabilities. To achieve this, we leverage algorithmic parsing techniques to establish smart contract sequences, utilize reproducible stateful computation techniques, and incorporate innovative and customizable algorithmic feedback mechanisms. This paper identifies the gap in the security spectrum, the limitations of existing techniques, and explains how the Mamori's architecture navigates these challenges, facilitating a new standard in smart contract auditing.

1 Introduction

In 2023, losses exceeding 1.8 billion dollars were recorded across 751 incidents, with almost half of these losses attributable to private key compromises in 47 incidents (Certik, 2024). The remainder of the losses were due to the exploitation of *technical and economic* risks. Noteworthy incidents include Euler Finance, which lost 200 million dollars due to a logic flaw, and BonqDAO, which was compromised for 120 million dollars through oracle manipulation. These intimidating incidents are the *fundamental roadblocks to mass adoption.* Technical risks exploit vulnerabilities in code and infrastructure, whereas economic risks manipulate economic design of protocols, often exploiting them through multi-function calls and smart contract state manipulation. Countermeasures for detecting smart contract vulnerabilities (Qian et al., 2022) encompass static analyzers and dynamic analyzers (fuzzing).

The primary limitation of static analysers lies in their focus on identifying technical risks, with a notable inability to detect economic risks. Economic risks often arise from complex interactions among smart contract function calls. Even when potential minor issues, such as rounding errors or integer overflows, are identified in a single function, the seemingly workable function is often combined with a sequence of actions for exploitation [Refer to section 4.1].

For dynamic analysers, while they have shown promising progress, they suffer from the inherent issue of an extensive search space and a high setup cost. For example, Echidna, developed by Trail of Bits, uses property-based testing and requires the definition of precise properties that the contract should always uphold. The need for accurate and comprehensive properties to catch vulnerabilities significantly increases the setup cost for auditors.

[*]email: andymlee@mamori.xyz

[†]This version of the paper is a preprint and should be considered as a work in progress, not as a finalized publication. Please refrain from further dissemination or citation without consulting the author.

Mamori aims to address this critical space within the DeFi sector. It leverages an algorithmic parsing technique to set up smart contract sequences, utilizes reproducible statefulness computation techniques in the r-evm environment, and incorporates innovative and customizable algorithmic feedback mechanisms to efficiently reduce the false negative search and discover zero-day exploits. Recognizing that technical risks, such as integer overflow, require function calls, a comprehensive detection algorithm focused on multi-function calls can effectively cover both technical and economic risks.

Our goal is to bridge the false-negative gap. Even after undergoing a thorough audit, our algorithm is still able to uncover vulnerabilities in an efficient way. Mamori also identifies the commonality between traditional MEV and economic risk as a homogeneous optimization problem. Our algorithm, with customizable features, will be extended to serve as a path-finding layer, covering the role of an MEV searcher and maximizing its utilization.

This paper is structured as follows: Section 2 introduces our perspective on the root causes of smart contract security issues. Section 3 presents the Mamori's architecture and explains our approach to formulating economic risk. Section 4 demonstrates the feasibility of our innovative solutions with extrapolatable case studies. Section 5 concludes the discussion.

2 Root Cause of Vulnerabilities in Smart Contract Security

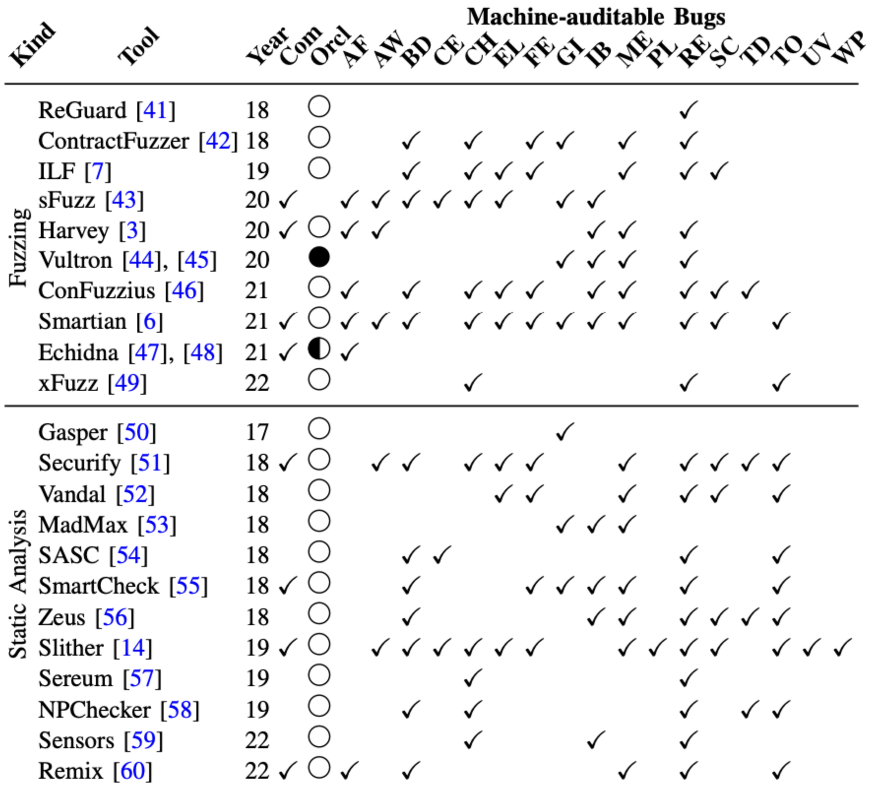
The current issue in security audits lies in the suboptimal approach to detecting the "known unknowns" region, which is publicly known by expert but the team is unaware of it. With the rising incidence of zero-day smart contract attacks, the primary emphasis should be on defending against attacks that are currently unknown, referred to as "unknown unknowns." However, security properties in static prevention solutions are closely tied to known attacks and vulnerabilities, and these properties may not provide comprehensive protection for smart contracts against new vulnerabilities (Ivanov et al., 2023).

There are two broad categories for detecting smart contract bugs: static analysis, which includes formal methods and symbolic execution, and dynamic analysis, commonly referred to as fuzzing. Static analysis involves program analysis techniques used to analyse code and programs without executing them. Security researchers identify potential bugs through intermediate representation-based analysis methods, using a set of predefined or user-specified specifications, and generate a human-readable security report. However, static analysers often suffer from issues such as overestimation of smart contracts, leading to high false positives, or precise enumeration of symbolic traces, resulting in excessive execution paths.

In dynamic analysis, there are two common approaches to fuzz testing: stateless and stateful fuzzing. Stateless fuzzing disregards the previous state and reuses the state at each run (e.g., UniTest), limiting its ability to detect state-related exploitations. Stateful fuzzing, also known as invariant testing, generates a test case, i.e., the action sequence, and mutates parameters to detect vulnerabilities. The challenge lies in the inherent difficulty of revealing bugs hidden in the 'exploitable regions' not covered by the fuzzer, which can result in false negatives (Qian et al., 2023).

Static analysis and even emerging machine learning technologies like LLM-based detection are constrained by past exploit experiences. Models and tools are trained based on historical exploitation data to detect relevant vulnerabilities from the database. However, economic vulnerabilities are often discovered under unique parameters and states with complex action sequences. Therefore, we need a more comprehensive tool to address the root problem, and stateful fuzzing, considering different parameters and combinations, should become the new standard for smart contract security.

2.1 Limitations of Current Analytical Tools



Kind	Tool	Year	Com	Orcl	Machine-auditable Bugs																
					AF	AW	BD	CE	CH	EL	FE	GI	IB	ME	PL	RE	SC	TD	TO	UV	WP
Fuzzing	ReGuard [41]	18		○												✓					
	ContractFuzzer [42]	18		○			✓		✓		✓	✓		✓		✓					
	ILF [7]	19		○			✓		✓	✓	✓			✓		✓	✓				
	sFuzz [43]	20	✓		✓	✓	✓	✓	✓	✓		✓	✓								
	Harvey [3]	20	✓	○	✓	✓							✓	✓		✓					
	Vultron [44], [45]	20		●								✓	✓	✓		✓					
	ConFuzzius [46]	21		○	✓		✓		✓	✓	✓		✓	✓		✓	✓	✓			
	Smartian [6]	21	✓	○	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓		✓		
	Echidna [47], [48]	21	✓	◐	✓																
	xFuzz [49]	22		○					✓							✓			✓		
Static Analysis	Gasper [50]	17		○								✓									
	Securify [51]	18	✓	○		✓	✓		✓	✓	✓			✓		✓	✓	✓	✓		
	Vandal [52]	18		○						✓	✓			✓		✓	✓		✓		
	MadMax [53]	18		○								✓	✓	✓							
	SASC [54]	18		○			✓	✓								✓			✓		
	SmartCheck [55]	18	✓	○			✓				✓	✓	✓	✓		✓			✓		
	Zeus [56]	18		○			✓						✓	✓		✓	✓	✓	✓		
	Slither [14]	19	✓	○		✓	✓	✓	✓	✓	✓			✓	✓	✓	✓		✓	✓	✓
	Sereum [57]	19		○					✓							✓					
	NPChecker [58]	19		○			✓		✓							✓		✓	✓		
	Sensors [59]	22		○					✓				✓			✓					
	Remix [60]	22	✓	○	✓		✓							✓		✓			✓		

Figure 1: Zhang et al., 2023. Demystifying Exploitable Bugs in Smart Contracts.

The table displays the current limitations of the existing techniques, with a focus on the "**Orcl**" column, representing test oracles. ○, ◐ , and ● correspond to fixed and simple oracles, hand-coded oracles, and oracles that can automatically adapt to cover a wide range of functional bugs, respectively.

The current techniques primarily rely on either simple and general oracles (denoted as ○) or project-specific hand-coded oracles (represented by ◐) (Zhang et al., 2023). *Echidna* (Grieco et al., 2020) requires a substantial hand-coded setup and a deep understanding of the targeted smart contract logic, while *Vultron* (Wang, 2019) is not a fuzzer but rather a general-purpose vulnerability detection oracle.

Most fuzzers involve simple invariant testing, while Mamori focuses on the ultimate and most crucial oracle test, which is an economic exploit, while providing customizable features for users to test specific invariants if needed. We argue that despite static analysers discovering a wide variety of bugs, attackers still need to call function(s) as an entry point for exploitations. Therefore, our ideology is to serve as the final guard to protect the gateway of exploitation with the most crucial invariant test and advanced algorithmic solutions.

2.2 Current Toolings for Economic Risk - Stateful Fuzzers

Fuzzing refers to the process of sending a set of random inputs into the program and stress-testing it against certain predefined conditions. It is also known as breaking the invariant. If the invariant does not hold under certain function calls, it is considered program bugs. To uncover complex vulnerabilities and bugs, a stateful fuzzer manipulates the program state across multiple operations and allows the fuzzer to explore deeper state-related bugs, such as those that manifest only after a series of specific actions. Mamori treats the ultimate goal of stateful fuzzing as finding economic exploits, which requires the testing to reach a unique state as a precondition and execute the inaccurate function to extract values.

To achieve this, in addition to the concept of reaching an exploitable state, our protocol defines three core implementation components. First, the action sequence, also known as the initial seed, test case, test scenario, or execution paths, represents the combinations of function calls aimed at revealing vulnerable execution paths in smart contracts. Each fuzzer may have its unique transaction ordering and dependency analysis to observe the function interactions on contract state and behavior.

Second, the fuzzing inputs, also known as input vectors or test data, consist of the input parameters of a function. These parameters are mutated across iterations and the testing process through a feedback mechanism. Customizable heuristics and anomaly detection rules are employed to identify suspicious behavior and guide the mutation of the next set of fuzzing parameters. In a traditional coverage-guided fuzzer, input mutations are performed randomly, and the fuzzer executes the mutated input to check whether the invariant is upheld.

The third core component of stateful fuzzing is the test oracle, which is commonly known as an invariant, test assertions, or security properties. The test oracle defines the vulnerability, and if the oracle is violated during testing, it indicates the presence of the targeted vulnerability.

Upon completing the stateful fuzzing process, coverage metrics are used to assess how thoroughly the fuzzer has tested the smart contract. However, it's important to note that having the highest code coverage does not necessarily guarantee the minimum chance of false negatives. Given the complex search space in stateful fuzzing, the literature has not reached a consensus on the best metrics to evaluate the performance of fuzzers. A combination of metrics, including code coverage, function coverage, input space coverage, and state coverage, may be used to ensure comprehensive testing.

Current stateful fuzzers primarily innovate in these three components. *CONTRACTFUZZER* (Jiang et al., 2018) was one of the earliest fuzzing frameworks, extracting the data types of each arguments of API functions and signature used in ABI functions. However, its testing process involves random function invocations with random inputs within the valid input domain. *HARVEY* (Wüstholz et al., 2020) fuzzes transaction sequences in a demand-driven manner, utilising regularity or aggressiveness based on coverage increase. It generates input parameters using predictive inputs, considering specified cost metrics such as executions that flip a branch condition to increase coverage. *sFUZZ* (Nguyen et al., 2020) introduces innovation by generating new test cases using adaptive objective function and AFL fuzzer, employing genetic algorithm, , and updating based on the feedback. Its aim is to create a set of test cases that maximises branch coverage . *ECHIDNA* (Grieco, 2020) is one of the few fuzzers that innovates through the test oracle. Auditors can set up properties involving the testing process of pre-conditions, actions, and post-conditions. They check if the combinations of transactions fulfill the expected outcomes. However, using *ECHIDNA* requires a high entry barrier, as understanding the smart contract and identifying relevant hand-coded properties may significantly increase the setup cost.

CONFUZZIUS (Torres et al., 2021) was the first fuzzer to employ a hybrid approach. It extracts data dependencies through static analysis and dynamically retrieves access patterns to state variables at runtime by iterating through the execution trace for opcode SLOAD and SSTORE instructions to generate action sequences. This hybrid approach was later extended by both *SMARTIAN* (Choi et al., 2021), which enhances the fuzzing module to mutate parameters based on DataFlowGain by identifying new functions that use any variable defined by previous calls to access new states, and

CROSSFUZZ (Yang et al., 2024), which generates action sequences based on inter-contract data flow information and dependencies.

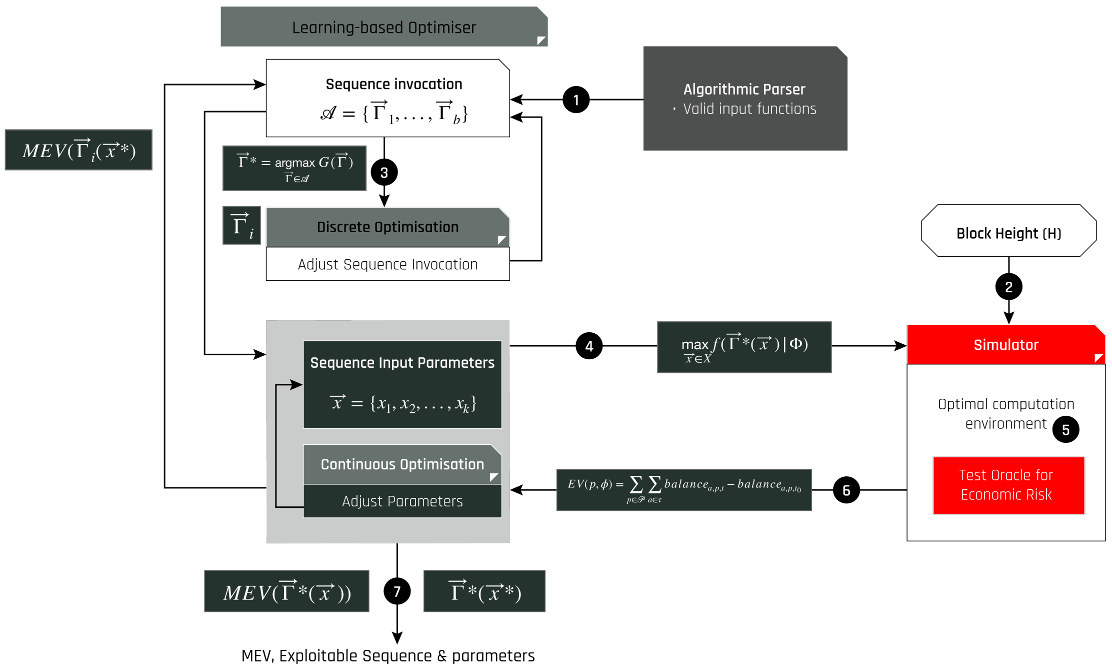
There are some more recent emerging fuzzers. *IR-FUZZ* (Liu et al., , 2023) extends *sFUZZ* on the test cases mutation by introducing techniques that consider data dependencies between functions to explore richer states. It also incorporates a branch distance-based measure and branch search algorithm for energy allocation. *ITYFUZZ*'s (Shou et al., 2023) core innovation lies on the snapshot-based approach which allows for quick backtracking to the interesting state, namely corpus, and re-execution of fuzzing with different action sequences toward interesting corpus.

Mamori, as a security protocol designed to minimize smart contract economic risks, identifies the current limitations of fuzzers. The core limitation is that most fuzzers combine test case generation with parameter mutations. The rationale behind this is to maximize certain metrics such as branch coverage, data flow information, control flow, and code coverage. Despite this methodology potentially increasing the number of tested action sequences, it may significantly increase the chance of false negatives since the exploitable region based on input parameters given a sequence is often narrow (Check 4.2). Moreover, most fuzzers test against a simple and general oracle (See 2.1) and are limited at the syntactical level, suffering from both false negatives and false positives. These oracles are not tested against economic risk but assume domain patterns for specific types of vulnerabilities.

The following sections will demonstrate how Mamori addresses these limitations with interdisciplinary insights and the latest technology, along with additional features to mitigate the core vulnerability issue - economic risk.

3 Methodology

3.1 Mamori's Architecture



Learning-based Optimiser
Sequence invocation
$\mathscr{A} = \{\overrightarrow{\Gamma}_1, \dots, \overrightarrow{\Gamma}_b\}$
1
Algorithmic Parser
· Valid input functions
$MEV(\overrightarrow{\Gamma}_i(\overrightarrow{x}^*))$
$\overrightarrow{\Gamma}^* = \operatorname{argmax}_{\overrightarrow{\Gamma} \in \mathscr{A}} G(\overrightarrow{\Gamma})$
3
$\overrightarrow{\Gamma}_i$
Discrete Optimisation
Adjust Sequence Invocation
Block Height (H)
2
Sequence Input Parameters
$\overrightarrow{x} = \{x_1, x_2, \dots, x_k\}$
4
$\max_{\overrightarrow{x} \in X} f(\overrightarrow{\Gamma}^*(\overrightarrow{x}) \mid \Phi)$
Simulator
Optimal computation environment
5
Test Oracle for Economic Risk
Continuous Optimisation
Adjust Parameters
$EV(p,\phi) = \sum_{p \in \mathscr{P}} \sum_{a \in \tau} balance_{a,p,t} - balance_{a,p,t_0}$
6
$MEV(\overrightarrow{\Gamma}^*(\overrightarrow{x}))$
7
$\overrightarrow{\Gamma}^*(\overrightarrow{x}^*)$
MEV, Exploitable Sequence & parameters

Our architecture is fine-grained through the Lanturn framework (Balbel et al, 2023) and we focus on the practicality on detecting economic risk.

1. We leverage algorithmic modelling on the source code to reduce all user callable functions $\mathcal{N}$ to state-changing functions N (❶)
2. The block height (h) (❷) pre-define the initial state of the blockchain environment. For pre-deployment auditing, we start with arbitrary state or empty state.
3. The test case formulation (❸) learns the optimal action sequence, $-^*$, from several sources such as discrete optimisation methods. The optimal action sequence iteratively learn the simulated outcome and learn the potential optimal action sequence. On top of our proposed methods, we also integrate other fuzzers specialising test case formulation for comprehensiveness. Mamori also provides customisable features on prioritising the more suspicious case based on historic data and auditor preference.
4. Given an action sequence, Γ_i, we conduct a black-box optimisation for the continuous search space (❹) at the simulator (See 3.6 (❺)), lower-level r-evm environment, and return the maximised value based on our test oracle (❻).
5. Ultimately, our system will return the trace of the exploitable action with their corresponding input and output(❼)

3.2 The Optimisation Problem of Stateful Fuzzing

The optimization in our system involves two parts: a bi-level optimization in a hierarchical structure with an upper-level for test case generation and a lower-level for input space exploitation. The first part involves optimizing discrete values within the action space. This can be computationally intensive, especially when testing all possible user actions with at most K actions. The goal is to eliminate sequences that are highly unlikely to have a positive exploitable value (EV).

The second part deals with the optimization of a non-convex black-box function in the absence of a dataset. The combination of action sequences creates a highly convoluted function for calculating the exploitable value. Depending on the discrete action lengths, each action sequence has its own unique form of convexity. Thus, we cannot generalize a clear mathematical form of the objective function since a change in one action within an action sequence could drastically alter its functional form.

Our unconstrained input parameters are mostly continuous, which exponentially increases the continuous search space. To address this issue, we must develop heuristics to constrain the initial input parameters, such as setting limits based on the depth of liquidity within the smart contract, known as the optimization bound. In addition to the challenge of unconstrained continuous input parameters, non-convex black-box optimization inherently exhibits high dimensionality, leading to increased computational costs. Moreover, this high dimensionality makes the optimization process sensitive to the choice of initial search parameters and hyperparameters.

3.3 Our Approach to Detect Economic Exploits

3.3.1 Settings

The exploitable value of any DeFi system can be defined as:

$$EV = f(\vec{\Gamma}|\Phi)$$

, where $\vec{\Gamma}$ is a vector representing an unique action sequence with a length of k containing user functions $\gamma_1, \gamma_2, ..., \gamma_k \in \mathcal{N}_{viable}$. The total set of user-callable functions is defined as $\mathcal{N}$, and our pre-screening process will identify a subset of viable functions $\mathcal{N}_{viable} \subset \mathcal{N}$ based on their impact on smart contract states, ϕ. Φ contains a vector of system states $\phi_1, \phi_2, ..., \phi_k$.

An attack usually takes more than 1 step and the system states will be affected by Γ and its previous value after each time step.

$$\phi_t = g(\gamma_{t-1}|\phi_{t-1})$$

After establishing the set of action sequences, for each sequence Γ_i within the total viable set $\mathcal{A}$ of distinct action sequences, we optimise the user input values $\vec{x} = (x_1, x_2, ..., x_k)$ for each function γ_i in Γ_i.

3.3.2 Optimization Process

We can express our optimisation process in a bi-level manner, with the higher level being the discrete optimisation, which can be formulated as follows:

$$\vec{\Gamma}^* = \underset{\vec{\Gamma} \in \mathcal{A}}{\operatorname{argmax}}\ G(\vec{\Gamma}) \tag{1}$$

where G is a function (possibly based on historical rewards or external matrics) that evaluates the potential of the maximising EV of each action sequence in $\mathcal{A}$.

Once $\vec{\Gamma}^*$ is selected, we proceed with a lower-level continuous optimisation over user input values $\vec{x}$ and can be express as

$$\max_{\vec{x} \in X} f(\vec{\Gamma}^*(\vec{x})|\Phi) \tag{2}$$

, where X is the valid choice for $\vec{x}$ that constrain the infinitely continuous search space. The ultimate goal of Mamori is to find the optimal $\vec{\Gamma}^*$ with the corresponding $\vec{x}$ that provide a positive EV.

3.4 Test Oracle for Economic Risk

The test oracle is also known as the invariant that we test against.. The current tools pursue a specific purpose of auditing, defining invariant at a syntactical level. For example, SMARTIAN test against 13 classes of test oracles such as assertion failure (AF) which corresponds to the execution of an INVALID instruction. Almost all vulnerability patterns at the syntactic level can be traced back semantically to the mismatch between the actual transferred amount and the amount reflected in the contract's internal bookkeeping (Wang et al., 2019). They propose two invariants: the balance invariant, based on the consistency between the contract's balance and the sum of all participants' bookkeeping balances, and the transaction invariant, based on the amount deducted from a contract's bookkeeping balances and its deposition into the recipient's account. Apart from their implicit challenges with the oracle, such as identifying bookkeeping variables, handling non-currency assets, and verifying invariants under gas consumption, we argue that these invariants also focus on the inconsistency of smart contract functions and precision errors rather than economic risk.

Therefore, our view of economic risk is based on whether there is an increase in the attacker's balance. An economic exploit occurs if and only if the balance of an attacker after the action sequence is greater than the post-action balance. This concept shares commonality with the Maximal Extractable Value (MEV) derived from arbitrage opportunities and market conditions (Babel et al., 2023).

Therefore, we define the invariant of the economic risk as follows:

$$EV(p, \phi) = \sum_{p \in \mathcal{P}} \sum_{a \in \tau} balance_{a,p,t} - balance_{a,p,t_0} \tag{3}$$

, where p is address used during the exploit and $\mathcal{P}$ is the total number of addresses. Some exploits involve the creation of multiple address such as liquidator and borrower. τ is the total type of token and we compare the balance a. t refers to the time and t_0 refers to the initial time. This test oracle for economic risk is generalised at the semantically level and can be understood as given any state, ϕ, the total of the post transaction exceeds the initial balance.

3.5 Sequence Invocation

To obtain significant efficacy gain, we must find a way prioritize function-call sequences with a high probability to trigger an economic exploit. To do so, we deploy heuristics in Mamori to help identify high-value sequences, with high-value being defined as a high probability of a given sequence containing value extraction opportunity. Heuristics involved here are:

- If a given function f(x) reads and make use of state, ϕ, and function g(x) contains the logic that will alter state, ϕ, then g(x) can be executed before f(x)
- By "make use of", it could mean the following:
  - require statements that needs ϕ to be at certain state before f(x) could continue execute
  - if-else statements that determines which branch f(x) will proceed on given ϕ

  Taking a function sequence as an example, "*function deposit()* $\rightarrow$ *withdraw()* $\rightarrow$ *swap()*" would have a higher value than "*function withdraw()* $\rightarrow$ *deposit()*" because intuitively, if a deposit is made by a user, then there would be a balance state altered, and that would enable the ability of withdrawal.

To further simplify and illustrate, below is a Solidity Snippet:

```
// SPDX-License-Identifier: Unlicensed
pragma solidity ^0.8.0;

contract SimpleStorage {
    uint256 public storeIndex;
    mapping(address => uint256) public balances;

    function set(uint256 newIndex) public {
        // Change the state
        storeIndex = newIndex;
    }

    function updateBalance() public {
        // Use the state for arithmetic
        balances[msg.sender] = storeIndex * 2;
    }
}
```

Alongside is the pseudocode that outputs sequence that should be prioritized:

Algorithm 1: Output Sequence Prioritization

```
function FindStateDependencies(SolidityContract):
    ContractFunctions, States ← ExtractFuncAndState(SolidityContract);
    FunctionOrder ← ∅;
    for State in States do
        FunctionOrder.push(FindFuncThatAccessState(State));
        FunctionOrder.push(FindFuncThatWriteState(State));
    end
    return FunctionOrder;
```

In the above Solidity snippet, '*FunctionOrder*' sequence would therefore be *[updateBalance(), set()]*. With the '*FunctionOrder*' outputted, the Mamori execution module can then first execute functions at the top of the stack (*'set()'*), and execute function at the bottom of the stack the last *('updateBalance()')*.

Therefore, the heuristic of sequence invocation has the ability to prioritise interrelated functions. Function sequences that are interrelated will feed into the optimizer first and test for economic vulnerabilities given changing and using certain global states throughout the execution.

3.6 Stateful Computation

One of our core features in the realm of stateful fuzzing is the utilization of past input results as guidance for future inputs in order to gradually locate vulnerabilities. This section gives into our implementation with a specific focus on optimizing computational speed.

In pursuit of maximising test velocity, we begin by examining the Ethereum instance at its most granular level. Our fundamental question is: "How can we streamline the Ethereum instance by removing all unnecessary components and retaining only those essential for the seamless operation of our product?"

Ethereum, as a blockchain, operates as a continuous and stateful environment that persists indefinitely. However, for Mamori, we only require an execution environment tasked with input validation, output analysis, and iterative input optimization. As such, our requirements can be met with an ephemeral execution environment with the usage of a temporary database residing in memory. The temporary or in-memory database would have a copy of all relevant key-value pairs from the blockchain as context for running Mamori. Therefore, Mamori is not only capable of testing pre-production smart contracts, but also smart contracts that are already live and serving customers in production blockchain networks.

To establish an ephemeral execution environment, we initially began with Foundry, a testing toolkit, but eventually opted to directly utilize revm, the Rust implementation of the EVM. Foundry served as an accessible starting point with its integration of the Rust implementation of the EVM (revm). Additionally, Foundry offered flexibility, including adjustable block gas limits and contract sizes, enabling us to execute a substantial number of iterations with the algorithm.

However, tests in Foundry are written in Solidity, meaning that support for complex mathematics is limited, making it challenging to implement intricate optimizing algorithms. Consequently, we chose to implement Mamori directly with the Rust implementation of the EVM, which allows us to achieve greater low-level control. This includes the ability to lock the test environment to a specific state with lower memory usage than Foundry and only use the temporary database, thereby enhancing computation speed. A few runtime optimization approaches we could take to substantially increase testing speed is:

- Removal of unneeded overhead in the execution environment
- Make the temporary database support asynchronous i/o
- Parallel execution

4 Proof of Concept

To assess the effectiveness of our approach, we meticulously document three extrapolatable cases in the past to facilitate the understanding of Mamori. At the current proof-of-concept stage, our primary objective is to demonstrate the effectiveness of our search algorithm, which is currently based on Swarm Intelligence—a field within artificial intelligence (AI) that leverages the collective behavior of elements in decentralized and self-organized systems (Tang et al., 2021). This approach enables more effective searches in high-dimensional, non-convex spaces.

While we are currently demonstrating the cases with Swarm Intelligence, we remain open to the possibility of exploring other machine learning algorithms or combinations thereof in our ongoing efforts to enhance the efficiency of our search methodology, tailored to the nature of the targeted protocol.

In our proof of concept (POC), we utilize Particle Swarm Optimization (PSO), which is a subset of Swarm Intelligence, to assess the feasibility of integrating interdisciplinary intelligence. PSO is initialized with a group of random (heuristic) particles, representing a random valid solution space. Each particle updates two properties during the iterative process: velocity and position. They individually search for the optimal Exploitable Value (EV) in their respective spaces and communicate iteratively with their individual optimal values. We then determine the global optimal solution based on the best of the current individual optimal values. The particles continuously adjust their velocities and positions based on both individual and global optimal solutions, and this process can be formulated as follows:

$$v_{t+1} = c_1 v_t + c_2 r_1 (p_{i,t} x_t) + c_3 r_2 (p_{g,t} x_t)$$

$$x_{t+1} = x_t + v_{t+1}.$$

v_t and x_t are the velocity and position of particle i and iteration t respectively. $p_{i,t}$ and $p_{g,t}$ are the individual and global optimal value of particle i found before iteration t. c_1, c_2, c_3 are the inertia, cognitive and social coefficients respectively and r_1, r_2 are the random parameters to facilitate exploration. The following sub-section explains the core insights from past exploited reproduced by Mamori's algorithm.

4.1 Raft Exploit - Denote Inflation. Lost: 3.2 Million

The economic exploit of Raft serves as an illustrative example highlighting the current limitations of smart contract security within the DeFi space. Raft is a lending protocol that enables users to generate R tokens by depositing Liquid Staking Tokens (LSDs) as collateral. The security issue arises in the "mint" function, which utilizes the *"divUP"* function for rounding up the minting calculation. An attacker can exploit the rounding-up behavior of the *"divUP"* function by making extensive donations of cbETH to artificially inflate the Raft collateral *"storedIndex."* As a result, the attacker can accumulate an excessive amount of collateral tokens due to the miscalculation. The attack is completed by minting and selling R tokens, demonstrating the vulnerability in the system.

Trail of bits (2023)'s audit reports suggested a satisfactory performance of arithmetic category and unable to spot the logical and design error of calculation function used in the mint function. An independent auditor, Antonio Viggiano (2023), later found the potential issue of the rounding error and suggested to round down on mint based on the principle of rule-of-thumb on rounding (Schaffranek, 2023).

This incident highlights two core security limitations: the imperfection of static analysis and the challenge of discovering exploitation even in the presence of potential errors. Dynamic analysis should theoretically cover this type of exploit, and Trail of Bits also suggested conducting dynamic and end-to-end testing of the system using Echidna in their audit report for Raft. Mamori aims to redefine the crypto security space by addressing these limitations with a focus on detecting economic exploits and setting a new standard for auditing with a focus on invariant testing.

In our proof-of-concept stage for this case, we have demonstrated two significant feasibility aspects for dynamic analysis in general. First, multi-block exploitation can be reduced to a single block. In hindsight, the attack manipulated the *'currentIndex'* (the key state that trigger rounding errors when significantly moved) through multiple donations and liquidations, and our POC showed that this is reducible. Second, certain types of exploitation involving loops can also be reduced to a single action. The attack repeated the minting action sixty times in a loop. However, with Mamori's oracle for economic exploits, we can already detect the exploitative sequence with a single action.

Our proof-of-concept further demonstrates that while an real-world attack might be significantly complex that are difficult to analyse, which include steps such as creating a suitable attack environment and maximizing exploitable value, they can in fact be all abstracted into exploits of flawed protocol design, which Mamori is exactly good at - abstracting complexity and locating what's hidden.

4.2 Saddle Finance - Logic Flaw. Lost: 10 Million

Saddle Finance is an AMM that uses elements of Curve within its protocol. But instead of forking the original Vyper code, they reimplement Curve's Meta pool in Solidity allowing a token to be swapped for an LP token. The error exists from the price miscalculations of the value of tokens given, based upon the base virtual price of the LP token. The bug is triggered from the function of 'MetaSwapUtils' allowing attack to extract economic profit by simply swapping sUSD for LP token followed by swapping LP token back for sUSD via Meta pool. The same bug also got exploited by its fork Synapse and Nerve which were drained over $8 million and $537,000 respectively. The same bug contributes total 18 million drained.

Saddle Finance found 3 auditing agencies, Certik, Quantstamp and OpenZepplin, to conduct their protocol audit. However, the cost of auditing is accounted by number of contracts and it is common that auditors do not audit those contracts which are out of their scope. Consequentially, despite Certik and OpenZeppelin have conducted a protocol audited and Quantstamp audited both protocol, virtual swap and token contracts, the contract related to Meta-pool swap is out of their scope. Thus, the function of an unaudited contract results in a huge exploitation.

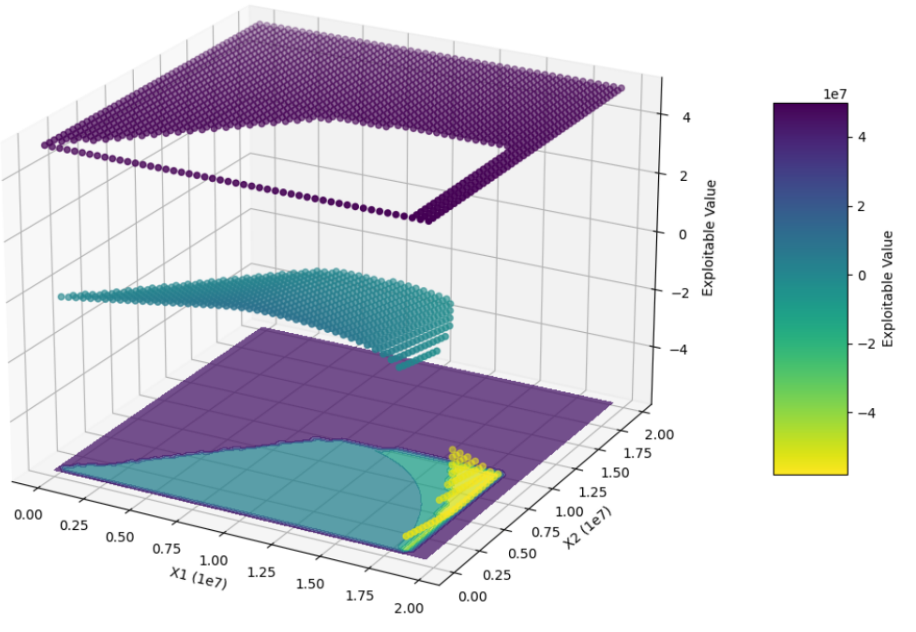


Figure 2: The exploitable continuous region of Saddle Finance Exploit

Our POC shows that although it is a rather simple exploitation with 2 dimensional step, the exploitable region is still relatively narrow which requires an intellectual exploration of the continuous space.

We found the maximise exploitable value and visualise the 2 dimensional exploitation for deeper understanding. Figure 1 shows the extensive continuous search space of the exploitable region. x_1 and x_2 refer to the parameters of input of swapping to LP token and swapping back for sUSD respectively. Purple region refers to the invalid input parameters space and the green region refers to the valid input parameters space without exploitation. The yellow region refers to the combinations of x_1 and x_2 to successfully execute the economic exploit.

4.3 Deus DEL - Wrong implementation. Lost: 5.4 Million

DEUS Finance is a platform for decentralized financial services and back in May 2023, their stablecoin *"DEIstablecoin"* executes an incorrect burnFrom function. Instead of using the OpenZeppelin ERC20 standard, the protocol implements their own burnFrom function but their *"currentAllowance"* is written as *"_allowances[account][_msg.sender()]*, instead of *"_allowances[_msg.sender()][account]"*, ultimately using the victim's tokens without authorisation.

Similarly to Saddle Finance, Certik audited the AMM product of the Deus Finance which is out of scope of the Stablecoin vulnerability. Although Armor Labs conducted audit on the lending protocol, but the issue contract *"LERC20Upgradable.sol"* is not included in the audit report. The purpose of this POC case is to demonstrate the ability of Mamori's integration on detecting the economic risk of self-written implementation.

5 Conclusion

Existing tools are far from perfect, and we need a more comprehensive platform to enhance smart contract security. Some features that are necessary for auditing, such as multi-contract auditing, are crucial. Zhang et al. (2023) have shown that price oracle manipulation is the most commonly exploited vulnerability across various popular DeFi categories, including Dexes, Yields, Derivatives, and yield aggregators, while lending is susceptible to implementation-specific bugs. xFuzz (Xue, 2022) has revealed that CONTRACTFUZZER and sFUZZ may miss some cross-contract vulnerabilities, whereas xFuzz focuses on cross-contract vulnerabilities. However, their machine learning-guided fuzzing may be weak in optimizing the continuous search space and may not cover certain issues like integer overflows/underflows (Qian et al., 2023).

Mamori identifies the existing limitations of security auditing techniques. With its innovative framework that improves on many details, including:

- An advanced stateful computation structure.
- Reduction of the setup cost in finding valid functions with algorithmic parsing implementation.
- An innovative interdisciplinary feedback mechanism and sequence initialization.
- A customisable feature to act as an MEV searcher.

Existing techniques, including ML models and static analysers, have progressively covered the "known unknowns" region and attempted to eliminate recurrent exploits. However, we believe that no single algorithm can perfectly cover all types of economic exploits. Mamori's mission is to leverage diverse technologies and protect against the "unknown unknowns" vulunerabilities, providing a facilitative environment for innovative features in the smart contract space.

6 Reference

Antonio Viggiano, 2023. Review of raft-fi/contracts v1.0..master Report. Access at: Link

Certik, 2024. Hack3d: The Web3 Security Report 2023. Access at: Link

Choi, J., Kim, D., Kim, S., Grieco, G., Groce, A. and Cha, S.K., 2021, November. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In*2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*(pp. 227-239). IEEE.

Deus Finance, 2022. Lending audit. Armor Labs Audit Report. Access at: Link

Grieco, G., Song, W., Cygan, A., Feist, J. and Groce, A., 2020, July. Echidna: effective, usable, and fast fuzzing for smart contracts. In*Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*(pp. 557-560).

Ivanov, N., Li, C., Yan, Q., Sun, Z., Cao, Z. and Luo, X., 2023. Security Threat Mitigation For Smart Contracts: A Comprehensive Survey.*ACM Computing Surveys*.

Jiang, B., Liu, Y. and Chan, W.K., 2018, September. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In*Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*(pp. 259-269).

Nguyen, T.D., Pham, L.H., Sun, J., Lin, Y. and Minh, Q.T., 2020, June. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In*Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*(pp. 778-788).

Liu, Z., Qian, P., Yang, J., Liu, L., Xu, X., He, Q. and Zhang, X., 2023. Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting. *IEEE Transactions on Information Forensics and Security*,*18*, pp.1237-1251.

Saddle Finance, 2023. Audit Report. Access at: Link

Shou, C., Tan, S. and Sen, K., 2023, July. Ityfuzz: Snapshot-based fuzzer for smart contract. In*Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*(pp. 322-333).

Qian, P., Cao, R., Li, W., Li, M., Zhang, L., Chen, J. and He, Q., 2023. Empirical Review of Smart Contract and DeFi Security: Vulnerability Detection and Automated Repair.*arXiv preprint arXiv:2309.02391*.

Qian, P., Liu, Z., He, Q., Huang, B., Tian, D. and Wang, X., 2022. Smart contract vulnerability detection technique: A survey. *arXiv preprint arXiv:2209.05872*.

Qian, P., Wu, H., Du, Z., Vural, T., Rong, D., Cao, Z., Zhang, L., Wang, Y., Chen, J. and He, Q., 2023. MuFuzz: Sequence-Aware Mutation and Seed Mask Guidance for Blockchain Smart Contract Fuzzing. *arXiv preprint arXiv:2312.04512*.

Raoul Schaffranek, 2023. Tackling Rounding Errors with Precision Analysis by Raoul Schaffranek. Devcon Bogatá, Ethereum Foundation. Access at: Link

Tang, J., Liu, G. and Pan, Q., 2021. A review on representative swarm intelligence algorithms for solving optimization problems: Applications and trends.*IEEE/CAA Journal of Automatica Sinica*,*8*(10), pp.1627-1643.

Torres, C.F., Iannillo, A.K., Gervais, A. and State, R., 2021, September. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In*2021 IEEE European Symposium on Security and Privacy (EuroS&P)*(pp. 103-119). IEEE.

Trail of bits, 2023. Raft security review. Access at: Link

Wang, H., Li, Y., Lin, S.W., Ma, L. and Liu, Y., 2019, May. Vultron: catching vulnerable smart contracts once and for all. In*2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*(pp. 1-4). IEEE.

Wüstholz, V. and Christakis, M., 2020, November. Harvey: A greybox fuzzer for smart contracts. In*Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*(pp. 1398-1409).

Xue, Y., Ye, J., Zhang, W., Sun, J., Ma, L., Wang, H. and Zhao, J., 2022. xfuzz: Machine learning guided cross-contract fuzzing.*IEEE Transactions on Dependable and Secure Computing*.

Yang, H., Gu, X., Chen, X., Zheng, L. and Cui, Z., 2024. CrossFuzz: Cross-Contract Fuzzing for Smart Contract Vulnerability Detection.*Science of Computer Programming*, p.103076.

Zhang, Z., Zhang, B., Xu, W. and Lin, Z., 2023. Demystifying Exploitable Bugs in Smart Contracts. ICSE.