Lanturn: Measuring Economic Security of Smart Contracts Through Adaptive Learning

Kushal Babel*
Cornell Tech, IC3

Mojan Javaheripi*
University of California San Diego

Yan Ji
Cornell Tech, IC3

Mahimna Kelkar
Cornell Tech, IC3

Farinaz Koushanfar
University of California San Diego

Ari Juels
Cornell Tech, IC3

ABSTRACT

We introduce Lanturn: a general purpose adaptive learning-based framework for measuring the cryptoeconomic security of *composed decentralized-finance (DeFi) smart-contracts*. Lanturn discovers strategies comprising of concrete transactions for extracting economic value from smart contracts interacting with a particular transaction environment. We formulate the strategy discovery as a black-box optimization problem and leverage a novel adaptive learning-based algorithm to address it.

Lanturn features three key properties. First, it needs no *contract-specific* heuristics or reasoning, due to our black-box formulation of cryptoeconomic security. Second, it utilizes a simulation framework that operates natively on blockchain state and smart contract machine code, such that transactions returned by Lanturn's learning-based optimization engine can be *executed on-chain without modification*. Finally, Lanturn is *scalable* in that it can explore strategies comprising a large number of transactions that can be reordered or subject to insertion of new transactions.

We evaluate Lanturn on the historical data of the biggest and most active DeFi Applications: Sushiswap, UniswapV2, UniswapV3, and AaveV2. Our results show that Lanturn not only rediscovers existing, well-known strategies for extracting value from smart contracts, but also discovers new strategies that are previously undocumented. Lanturn also consistently discovers higher value than evidenced in the wild, surpassing a natural baseline computed using value extracted by bots and other strategic agents.

1 INTRODUCTION

The decentralized finance (DeFi) ecosystem, which consists of smart-contract-based financial protocols on blockchains like Ethereum, currently holds over $30 billion of stored value and executes billions of dollars worth of transactions every day [2]. DeFi contracts have enabled a wide array of financial blockchain applications including exchanging cryptocurrency tokens [14], margin and derivatives trading [7, 8], and collateralized lending [31]. DeFi has also witnessed the creation of novel financial instruments such as automated market makers [14], atomic swaps [38], and flash loans [35]. These instruments have no traditional-finance analog, and can only exist due to the *deterministic* and *atomic* execution via blockchains.

Economic security of smart contracts. The high economic value at stake in DeFi contracts has naturally attracted the attention of attackers. Observed attacks on smart contracts can be divided into two broad categories: (1) Traditional software exploits and (2) Financial (Cryptoeconomic) exploits.

*The first two authors contributed equally to this work.

Many notable historical attacks on DeFi contracts, such as, reentrancy [30], batch overflow [23], incorrect initialization [5] can be categorized as traditional software exploits. Existing tools and well-studied techniques from the literature(e.g. [19, 28, 33]) can be squarely applied to analyze and mitigate this category of exploits.

On the other hand, there has been a rise in attacks (e.g. sandwich attacks [45], generalized frontrunning [37], liquidity sniping [40]) and strategic behavior (e.g. arbitrage [44], backrunning [34]) which are primarily financial in nature. This *cryptoeconomic* category of exploits is uniquely facilitated by blockchain systems and relies primarily on an adversary's ability to reorder, censor, and insert new blockchain transactions—either through strategic increase in transaction-fee payments or by using the power of block creators.

In this work, we develop Lanturn, which allows users, DeFi developers, and researchers to study the cryptoeconomic security of smart contracts, i.e., their exposure to financial attacks. At a very high level, Lanturn takes in a given set of concrete transactions, as well as any symbolic transactions with unknown parameters, and utilizes a novel adaptive learning algorithm to output an ordered sequence of concrete transactions that can be executed *on chain without modification* to extract economic value. In the remainder of this section, we first highlight why analysis of cryptoeconomic security is challenging, then give a perspective on two lines of work in literature that seek to address these challenges. Finally, we present an overview of Lanturn and our contributions.

DeFi composability and attacks. The co-existence of a plethora of DeFi contracts leads to interesting interactions, as the operation or state of one contract can in many cases influence those in other contracts. While *composition* of smart contracts is crucial for building interesting DeFi applications (e.g. collateralized lending [31]), it raises significant challenges in analyzing their cryptoeconomic security [16]. Security analysis of smart contracts seldom considers this kind of composability, often due to the fact that predicting the extent of interactions with future contracts is impossible [16]. As DeFi has grown, more complex and intricate financial interactions have been observed among multiple DeFi contracts whose composition is currently poorly understood.

Composability issues have been identified as the root cause of a number of high-profile DeFi attacks [35] in which attackers have cleverly exploited complex DeFi interactions and stolen billions of dollars in the aggregate. As an example, flash-loans, which can provide virtually unlimited capital as long as the loan amount is returned within the same transaction, have been used to attack contracts designed without anticipation of sudden injections of capital. Crucially, these attacks target *economic security* in contrast to more traditional security exploits that take advantage of, e.g., programming errors or network vulnerabilities. The attacker's actions often abide by the rules intentionally expressed in the contract's code.

To design DeFi contracts that resist such attacks, it is critical to gain an early understanding of their cryptoeconomic security both in isolation and in composition with other contracts. A key notion reflecting cryptoeconomic security is *maximal (previously miner) extractable value* (MEV) [16, 20]. Intuitively, MEV quantifies cryptoeconomic security by expressing the maximum profit that can be extracted from a smart contract system. MEV encompasses all profit-seeking strategies and thus includes everything from commonplace arbitrage strategies and generalized frontrunning bots to complex composability exploits on, e.g., governance mechanisms.

Existing literature on DeFi economic security. Two broad approaches have emerged in the literature.

The first approach employs attack heuristics and hard-coded contract-specific strategies, and scans the transaction data and blockchain state essentially looking for patterns. This technique is highly performant and has found success at comprehensively finding simple MEV opportunities like arbitrage cycles [44] or sandwich attacks [45]. A major limitation however is the lack of generalizability; since the heuristics and hard-coded strategies typically come from exploits already seen in the wild, new strategies or worst-case attacks are seldom found; this results in a poor understanding of the overall risk. Furthermore, as new contracts are added, heuristics for their interaction with previous contracts need to be developed from scratch; this leads to more manual effort as DeFi ecosystems become more complex.

The second, more recently developed technique [16] forgoes prespecified strategies and directly models the semantics[1] of relevant smart contracts programmatically. This allows for generic path exploration to find all possible (including novel, previously unseen) strategies through the use of formal verification tools. It also allows analysis of the worst-case behavior, thereby providing concrete, *provable* upper bounds on the exploitable value of composed DeFi contracts. Unfortunately, as expected, a major drawback of this approach is the lack of scalability. There is an exponential blowup in the number of paths in terms of the number of transactions or the complexity of contract code. While clever parallelization or manual simplification of smart-contract models through, e.g., removal of irrelevant code, can help, scalability remains a major bottleneck.

The above discussion illustrates an all-too-common tradeoff between generalizability and scalability. The two existing techniques provide only one of these properties while significantly compromising on the other. This work bridges this gap by leveraging adaptive learning techniques. We propose Lanturn[2]: an efficient and generic tool to find MEV opportunities and understand the economic security risks of smart contracts and their composition.

1.1 Lanturn Overview and Contributions

Lanturn is a general purpose adaptive learning-based framework for measuring the cryptoeconomic security of composed DeFi smart contracts. As the validator (consensus node) represents the most privileged player in the ecosystem, the problem of cryptoeconomic security can be studied through the lens of Maximal Extractable Value (MEV) [16, 20] available to the validator. We now briefly present our problem formulation, the architecture and properties of Lanturn, and a summary of our contributions.

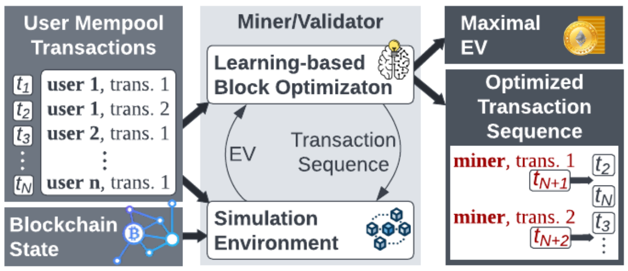


Figure 1: Lanturn problem formulation and architecture

Problem Formulation. The goal of an agent in the MEV landscape is to take as input a pending set of user transactions and output a new sequence of transactions—including some or all of the inputs—that maximizes the agent's EV (Extractable Value), i.e., profit. The output set constitutes some or all of a new block on the blockchain. Since the agent in Lanturn is by default a validator, it can maximize EV by any desired combination of reordering user transactions as well as inserting new transactions based on symbolic transactions with unknown parameters.[3] We do not consider explicit censoring of transactions — for the purposes of this work, censoring a transaction is no better than ordering it after the validator's last insertion. We assume that the agent holds a certain amount of starting capital in the form of the native cryptocurrency (such as ETH). We do not assume any additional permissions or powers available to the agent (e.g., stake in a governance protocol or participation in an oracle network [18]).

The core of Lanturn's architecture consists of two modules: An adaptive learning-based *optimization module* and a *simulation environment*. At a high level, the optimization module strives to learn an output transaction sequence that maximizes the agent's EV. It does so by querying the simulation environment as a black-box to evaluate candidate transaction sequences. Figure 1 depicts this formulation and Lanturn's architecture.

Properties. Lanturn has the following key properties:

(1) **Generalizability**: Lanturn does not require encoding attack heuristics or contract-specific strategies. It treats smart contract execution as a black-box simulation environment[4]. Moreover, Lanturn doesn't simplify smart contracts or translate them into another form. Its generalizability enables it to discover both known and new MEV-extraction strategies. Similar to [16], Lanturn does require symbolic templates for performing insertions. Specifying templates, hwoever,

[1]While the entire smart contract code can be faithfully simulated in theory, performance issues required [16] to manually remove the irrelevant code parts and encode only the core functionality.
[2]The light produced by the Pokémon Lanturn, nicknamed the "Deep-Sea Star," can illuminate even the darkest depths of DeFi.

[3]We do not consider transaction insertions that deploy new smart contracts with unknown code. The result would be an intractable problem space, and on-the-fly contract synthesis is rarely used in on-chain arbitrage today.
[4]While not a focus of this work, a Lanturn practitioner can easily integrate contract-specific heuristics to guide the optimization and find even more interesting strategies.

requires only a cursory knowledge of the functionality of the contract. It is not comparable in complexity to designing contract-specific strategies, which requires detailed understanding of the contract and more importantly, research and development for every new strategy.

(2) **Native execution**: Since Lanturn simulates actual smart contract bytecode, the obtained profit-yielding strategies are executable directly on-chain without any modification or weeding of false positives. In fact, our experiments are done directly on a private fork of the blockchain mainnet.

(3) **Scalability:** Lanturn scales with respect to both code complexity and search space size. Unlike prior work ([16, 44]), Lanturn does not use program analysis, so smart-contract-code complexity, typically arising due to loops (e.g. in Curve Finance [22], UniswapV3 [14]), recursions, etc. impose no additional overhead on Lanturn. Additionally, by leveraging adaptive learning, Lanturn can successfully handle vast search spaces (unlike prior work [16]) arising out of reordering of a large number of transactions (as many as 80) as well as insertion of new transactions with wide parameter ranges.

(4) **Adaptability to computation budget:** The Lanturn optimization algorithm can be tuned to match available computing resources and time budgets. Lanturn is also linearly parallelizable across servers in a straightforward way, further helping its scalability.

Contributions. Our work offers five concrete contributions:

- **Problem formulation as black-box optimization** (Section 4): We are the first to formulate the problem of quantifying the cryptoeconomic security of DeFi contract as a black-box optimization problem.
- **Adaptive-learning approach to MEV optimization** (Section 5.1): We present an adaptive-learning algorithm to scalably optimize MEV extraction and thus cryptoeconomic security measurement.
- **Lanturn tool** (Section 5.2): We develop the Lanturn tool which implements our learning-based optimization algorithm and a simulation environment which is tailored to efficiently simulate many transaction sequences on a private fork of the blockchain mainnet.
- **Experimental validation** (Section 6): We conduct extensive experiments with Lanturn on the biggest decentralized exchanges (Sushiswap, UniswapV2, UniswapV3) and one of the biggest lending protocols (AaveV2) using their real world activity from Ethereum mainnet. Our results show that Lanturn uncovers significant MEV—*surpassing a baseline derived from the value extracted by strategic agents in the wild.* In Figure 15, we also report that Lanturn uncovers significant MEV within tens of seconds to a few minutes for majority of the problem instances. For comparison, prior work [16] based on formal verification, takes more than an hour (under the same computation resources) to analyze a search space comprising of as little as 9 transactions.
- **Strategy discovery** (Section 6): Our experiments show that Lanturn not only re-discovers well-known profit-making MEV strategies but also *discovers new strategies* that are previously undocumented.

2 BACKGROUND

2.1 Decentralized Finance and Contracts

We refer the reader to Appendix A for a background on the mechanics of *constant function* Automated Market Makers (AMMs) such as UniswapV2, Sushiswap and UniswapV3 [13, 14], and Lending Contracts such as AaveV2 [12].

2.2 MEV

2.2.1 Maximal Extractable Value (MEV). Transaction ordering (i.e., the sequence in which transactions are executed) is critically important for DeFi applications. Since the block-proposing entity (miner in PoW and validator in PoS) is solely in charge of inclusion and ordering of transactions in the block that it creates, it can strategically leverage this power to capture value from users by e.g., reordering user transactions or inserting its own transactions. To capture such behavior, Daian et al. [20] coined the term *maximal* (previously *miner*) *extractable value* (MEV), which intuitively measures the profit extractable by the block-proposing entity. MEV encompasses a wide range of strategies including market-correcting forces like DEX arbitrage, simple adversarial behavior like frontrunning or sandwiching, as well as more complex, recent strategies; several recent works [34, 41, 44, 45] have attempted to quantify the total profit extracted for specific strategies.

To model arbitrary strategies, a concrete definition to quantify MEV (for Ethereum-like systems) was later formulated by Babel et al. [16]. We use this definition for our analyses, and provide the relevant formalism in Section 3.1.

Flashbots. While validators are in the prime position to extract MEV, they often do not have the expertise to find the optimal transaction ordering which gives them the maximal profit. This has led to the outsourcing the task of finding these "MEV opportunities" to specialized actors called *searchers.* Flashbots [9], an organization formed in late 2020, has emerged as the leader in the marketplace connecting searchers to validators. Informally, searchers find MEV opportunities and consequently bid for inclusion of their specific transaction sequences (or "bundles") inside the validator's block; here, Flashbots acts as a trusted intermediary ensuring that validators cannot steal the searcher's profits.

Since this MEV marketplace exists publicly, it allows us to estimate how much MEV was actually extracted on-chain. We will use this as the baseline for our Lanturn experiments. More relevant details of the inner workings of Flashbots and the impact of its design on MEV extraction are given in Section 6.

3 MODEL

3.1 MEV Formalism

We introduce some basic formalism based on Babel et al. [16].

System state. Let the space of all possible accounts (e.g., 160-bit identifiers in Ethereum) be denoted by $\mathcal{A}$. Note that $\mathcal{A}$ captures both user-owned and contract-owned accounts. For $a \in \mathcal{A}$, we use $\mathtt{balance}(a)[\mathbf{T}]$ to denote the balance of token $\mathbf{T}$ in account a, and $\mathtt{data}(a)$ to denote the other associated data such as smart contract code and storage. For simplicity, we use $\mathbf{T} = 0$ to denote the primary token (e.g., ETH in Ethereum).

Define the system state s as the combination of the balances and data associated with all accounts: $s(a) = (\text{balance}(a), \text{data}(a))$. Transactions are polynomial-sized (in the security parameter) strings constructed by some player that are executed by the system and can change the system state, similar to ACID-style database transactions [39]. Unlike traditional database transactions which perform simple read/write operations on the database, blockchain transactions can be executed dynamically according to the relevant smart contract code, which has Turing-complete semantics [42]. Note that even if the smart contract throws an error during this execution, the transaction still pays appropriate fees from the sender's account to the validator's account, and thus also modifies the blockchain state. These fees are based on the computation budget (called "gas") consumed by the transaction until the error is encountered.

A block in the blockchain, along with other protocol relevant data, contains a sequence of transactions. The block transforms the initial state to a new state, by executing the transaction sequence in order. Let action denote this transformation. Then, the execution of block B at initial state s can be represented by $s' = \text{action(B)(s)}$, where s' is the resulting state.

Extractable value. Extractable value (EV) in a state s for a player P is intuitively the maximum profit realizable by P in any state s' which is the result of some block B that can be constructed by P[5]. More formally, suppose that P controls a set A_p of accounts and let $\mathcal{B}$ denote the set of (valid) blocks that P can create in state s (this includes any transactions inserted by P along with those in the transaction mempool). Then, EV is defined as follows:

$$\text{EV}(P, s) = \max_{B \in \mathcal{B}} \left\{ \sum_{a \in A_P} \begin{array}{l} \text{balance}_{s'}(a)[0] \\ -\text{balance}_{s}(a)[0] \end{array} \right\}.$$

where $s' = \text{action}(B)(s)$. The *maximal* extractable value (or MEV) of a state s can now be defined as the maximum value of $\text{EV}(P, s)$ over all players P. Since the validator is strictly more privileged position than any other permissionless player in the system, MEV will typically be the EV corresponding to this entity.

3.2 Threat Model

The agent (validator) is given pending user transactions and the current blockchain state. It can reorder these transactions, but cannot modify the transactions themselves, as the transactions are cryptographically signed by the sender. The agent is given a fresh key-pair (and can generate additional ones) which can be used to create new transactions which interact with smart contracts or simply transfer cryptocurrency between accounts. Finally, for our experiments, we grant the account associated with the agent's given key-pair a certain amount of initial capital in terms of the native cryptocurrency (e.g. ETH).

4 LANTURN ARCHITECTURE

Figure 2 presents a high-level overview of the steps performed in Lanturn. In what follows, we explain Lanturn's execution flow, while referring to the component numbers in Figure 2.

[5]While Babel et al. [16] defines a more general version for multiple consecutive blocks, only single-block extensions are considered for their experiments.

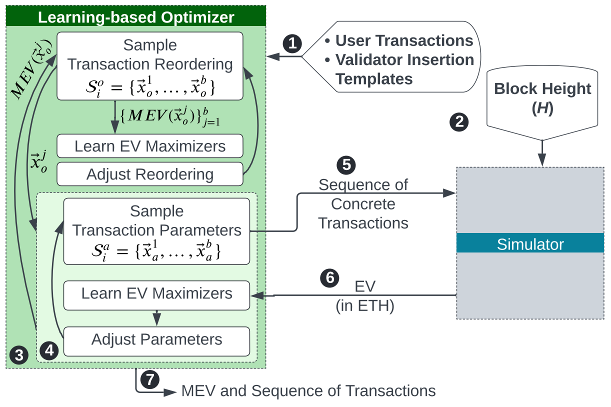

Learning-based Optimizer
Sample Transaction Reordering $S_i^o = \{\vec{x}_o^1, \dots, \vec{x}_o^b\}$
$\{MEV(\vec{x}_o^j)\}_{j=1}^b$
Learn EV Maximizers
Adjust Reordering
$MEV(\vec{x}_o^j)$
$\vec{x}_o^j$
Sample Transaction Parameters $S_i^a = \{\vec{x}_a^1, \dots, \vec{x}_a^b\}$
Learn EV Maximizers
Adjust Parameters
3
4
1
• User Transactions
• Validator Insertion Templates
Block Height (H)
2
5
Sequence of Concrete Transactions
Simulator
6
EV (in ETH)
7
MEV and Sequence of Transactions

Figure 2: Lanturn architecture and execution steps of the learning-based optimizer. Here, subscript i denotes the iteration number for the learning-based optimizer.

Lanturn consists of two interlinked components, namely a *learning-based optimizer* (*optimizer* for short) and a *simulator*. The optimizer takes as input a set of user transactions and template symbolic transactions that can be inserted by the validator (❶). An example of a validator insertion template can be abstractly represented as:

$$\texttt{swap}\ \ \alpha_0 \times \texttt{token}_0\ \ \texttt{with}\ \ \alpha_1 \times \texttt{token}_1,$$

where $\texttt{token}_0$ and $\texttt{token}_1$ are tokens interacted with in users' input transactions and α_0 and α_1 are what we call *template-variables*. What the template-variables denote depends on the template transaction – in this example, they may correspond to token amounts if the particular transaction denotes a token swap executed by the validator. Typically, template-variables denote the parameters of a certain public function of a particular smart contract (see Section 6 for details of template transactions and various examples). A concrete transaction sequence refers to an ordered list of user and validator transactions, wherein the template-variables inside the validator transactions are instantiated with concrete values of the appropriate type. The optimizer aims to determine the concrete transaction sequence such that its EV is maximized (❼). Each template transaction is used to derive exactly one concrete transaction. We provide details of our problem formulation for the optimizer in Section 4.1. As shown in Figure 2, our optimizer encompasses two hierarchical learning loops: the outer loop (❸) learns the optimal order of user and validator transactions while the inner loop (❹) learns optimal values for template-variables in a given sequence of transactions. Specifically, each transaction reordering in the outer loop (❸) initiates a call to the inner optimization loop, which then searches over the template variables (❹) and returns the MEV corresponding to the re-ordering. Each learning loop follows a carefully designed adaptive sampling algorithm outlined in Section 5.1.

For good performance, the optimizer relies on the simulator to provide accurate evaluations of EV for concrete transaction sequences. The simulator is given the state of the blockchain by specifying the block height H (❷), and the concrete transaction sequence from the optimizer (❺). The simulator returns the corresponding EV (❻). Details of our simulator are detailed in Section 5.2.

4.1 Problem Formulation

To estimate the MEV for a given set of user transactions and validator template insertions, we formulate the problem as a black-box optimization comprising three key components, namely, the optimization variables, the objective function, and the optimization constraints (bounds). We adaptively learn and explore the underlying space of values for the variables to find a solution, within the defined bounds, that maximizes the objective function. Since the search space is extensively large, linearly learning the whole space is not scalable. Our adaptive learning-based optimization is devised to be scalable by adaptively sampling and dynamically learning the important and eventful portions of the search space.

To simulate the interplay among various optimization variables, we concatenate them to form a vector $\vec{x} \in \mathbb{R}^d$. Here, d is the number of variables that shall be determined by the optimizer. The goal of black-box optimization is to solve the following problem:

$$\max_{\vec{x} \in X} F_1(\vec{x}). \tag{1}$$

Here $F_1(\cdot)$ is the optimization objective and X denotes the search space, which includes all the valid values for the variables based on the optimization constraints. We utilize EV as our objective function in Equation 1. The possible choices for the validator, i.e., the order of the transactions and the template-variables, represent the optimization variables $\vec{x}$. Figure 3 illustrates template-variables for an example problem. The constraints are set by the blockchain protocol and the context of the experiment: (1) if there are multiple transactions from a user, they should be ordered relative to each other according to a per-user serial number ("nonce") specified in each blockchain transaction; (2) the values for the template-variables are constrained by their type (such as the range of an integer datatype); (3) there can be further constraints in the context of a particular experiment, such as the validator's capital.

To optimize Equation 1 in the context of this work, we reformulate it as a bi-level optimization problem: First, a certain reordering of the transactions is proposed ($\vec{x_o}$); then, for the given transaction sequence, the optimizer seeks to find the concrete values of the template-variables ($\vec{x_a}$) that maximize EV. Using this formulation for bi-level optimization, Equation 1 can be rewritten as:

$$\begin{aligned} &\max_{\vec{x_o} \in X_o} F_1(\vec{x_o}), \\ &\text{where } F_1(\vec{x_o}) = \max_{\vec{x_a} \in X_a} F_2(\vec{x_o}, \vec{x_a}). \end{aligned} \tag{2}$$

Here, the lower-level objective function $F_2 : X_o \times X_a \longrightarrow \mathbb{R}$ represents the expected EV obtained from a concrete transaction sequence. X_o and X_a represent the underlying space of valid choices for $\vec{x_o}$ and $\vec{x_a}$ based on the aforementioned constraints. Specifically, due to the protocol's constraint on nonce, the space of transaction ordering X_0 is all permutations of user and validator transactions wherein all transactions from a particular user appear in the exact order specified by the user. The domain of template-variables is determined by the aforementioned constraints and can be further refined by the Lanturn practitioner (see Section 6).

Optimizing Equation 2 is especially challenging as the space of possible solutions grows exponentially with the number of optimization variables. Specifically, given $|\vec{x_o}|$ total transactions with n_i number of transactions from the i^{th} user and x_j possible values for the j^{th} template-variable, the number of possible combinations equals $\frac{|\vec{x_o}|!}{\prod_i n_i!} \times \prod_{j=1}^{|\vec{x_a}|}(x_j)$. As an example, let us consider the small toy problem in Figure 3 with only 4 transactions. The optimization space for this problem contains almost $3 \times 10^{10} = (4! \times 3 \times 500 \times 500 \times 1600)$ possible solutions, which is obviously impractical for brute-force evaluation. We therefore propose an improved optimizer in the next section that efficiently solves the problem in Equation 2 by leveraging adaptive learning techniques.

User Transactions:

- User1 swaps 100 `usdc` for `eth`
- User2 swaps 4 `eth` for `usdc`

Template Symbolic Transactions:

- Validator adds α_0 liquidity units in price range α_1 to α_2
- validator swaps α_3 `usdc` for `eth`

Template-variables:

α_0 : `uint128`; α_1, α_2 : `int24`; α_3 : `uint256`

Constraints:

$\alpha_0 \in \{1, 10, 100\}$; $1000 < \alpha_1 < 1500$;
$2000 < \alpha_2 < 2500$; $400 < \alpha_3 < 2000$

Figure 3: An example input to the optimization platform including a list of user and validator transactions with unknown variables (α_i). Lanturn adaptively learns the best reordering of transactions along with the corresponding transaction amounts that maximize the MEV.

4.2 Applications of Lanturn

Lanturn empowers various stakeholders in the ecosystem:

- Developers and researchers: Lanturn directly enables smart contract developers to understand the cryptoeconomic behavior of their smart contracts in isolation as well as when composed with other DeFi contracts. Blockchain researchers can utilize Lanturn to understand the impact of MEV on the (in)stability of a consensus mechanism [20].
- Users: Users can use Lanturn to learn the value exposed in their transactions. The methodology in Lanturn can be extended to also discover unknown transaction parameters inside user-provided transaction templates such that MEV extracted from users is minimized (similar to minimax optimization [21]). We leave this extension to future work.
- Strategic players: Lanturn can be used offline by strategic agents such as "MEV bots" to analyze strategies that extract economic value from users and smart contracts. Inter-block times (12 seconds for Ethereum) is usually too short for our single-server implementation of Lanturn in majority of the cases. For instance, Figure 15 shows that given 10 seconds, Lanturn uncovers $\approx 40\%$ of the final MEV in $\approx 25\%$ cases. However, we discuss in Section 6.4 that the inherent parallelism in Lanturn can be exploited to use Lanturn effectively in real-time[6].

[6]Supporting "MEV bots" is not the focus of this work

5 LANTURN METHODOLOGY

5.1 Adaptive Learning

Deriving a closed-form relationship between MEV and a set of input transactions, in terms of the template-variables appearing in the validator transactions, is extremely difficult even for simple smart contracts, and infeasible for general smart contracts. As explained above, Lanturn aims to find an optimal solution through sampled evaluations of the underlying optimization space. In this context, each sample represents one concrete transaction sequence (a specific permutation of transactions as well as concrete values for any template-variables). Recall that the Lanturn simulator executes a given concrete transaction sequence on a particular state and returns the corresponding EV. A uniformly random sampling of the space is unlikely to find the maximum EV with a limited sampling budget, especially as the size of the search space increases. Lanturn therefore employs an adaptive non-uniform sampler that gradually learns the objective function and automatically guides future samples towards parts of the search space where the EV is expected to be high.

Algorithm 1 Overview of Lanturn Sampling

```
Inputs: arbitrary objective function $F(\cdot)$, $X$, $N$, $N_s$, $b$
Outputs: $\vec{x}^*$
 1: $\mathcal{S} = \emptyset$, $F^* = 0$
 2: $\mathcal{P}$ = Uniform distribution
 3: for i=1 to $N$ do
 4:     sample $\mathcal{S}_i \sim \mathcal{P}(X)$, $|\mathcal{S}_i| = b$          ▷ Sample a new batch
 5:     $\mathcal{S} = \mathcal{S} \cup \mathcal{S}_i$
 6:     $F^* = \max_{\vec{x} \in \mathcal{S}} F(\vec{x})$                          ▷ Find current maximum
 7:     $\gamma_i = 1$, $\mathcal{S}_g = \emptyset$
 8:     while $|\mathcal{S}_g| < b$ do                                  ▷ Tune $\gamma_i$ and find $\mathcal{S}_g$
 9:         reduce $\gamma_i$
10:         $\mathcal{S}_g = \{\vec{x} \in \mathcal{S} | F(\vec{x}) > \gamma_i F^*\}$
11:     update $\mathcal{P}$ based on $\mathcal{S}_g$                       ▷ Update sampling strategy
12:     if $F^*$ did not change for $N_s$ consecutive steps then
13:         break                                                ▷ activate early-stopping
14: $\vec{x}^* = \operatorname{argmax}_{\vec{x} \in \mathcal{S}} F(\vec{x})$
15: return $\vec{x}^*$
```

Algorithm 1 is a pseudocode sketch of the adaptive sampling procedure in Lanturn. It takes as input the EV evaluator $F(\cdot)$, the underlying search space X, number of sampling iterations N, early-stopping tolerance N_s, and a sampling batch size b. In each iteration, a new batch of i.i.d. samples $\mathcal{S}_i$ is generated and evaluated. The current maximum EV, denoted by F^*, is then found across all evaluated samples. The sampling procedure adaptively changes its sampling distributions $\mathcal{P}$ to achieve a good probability of finding a near-optimal solution. To this end, it utilizes the best samples evaluated so far, dubbed the *good samples* $\mathcal{S}_g$, to learn and update $\mathcal{P}$.

A proximity parameter $\gamma_i \in [0, 1]$ serves to select the current set of good samples $\mathcal{S}_g$, i.e., those close to the current maximum: $F(\vec{x}_g) \geq \gamma_i F^*$ ($\forall \vec{x}_g \in \mathcal{S}_g$). A large value of γ_i encourages exploitation of previously observed samples with a high EV while a lower value leads to more exploration. As such, we adaptively tune the value of γ_i by assigning it the maximum value $\in [0, 1]$ such that at least one batch of samples are marked as good. This, in turn, ensures the good samples $\mathcal{S}_g$ contain a diverse set of configurations, thereby balancing exploration and exploitation. Finally, using the derived $\mathcal{S}_g$, the sampling distributions are updated for the next iteration. This process continues until convergence to a (local) optimum, i.e., when the maximum found EV does not change for N_s consecutive steps, or until a maximum number of N iterations. The algorithm returns the maximizer $\vec{x}^*$ which corresponds to a concrete transaction sequence leading to the maximal EV (MEV).

Sampling distribution. Our sampling distribution $\mathcal{P}$ represents a combination of two distinct probability distributions that together facilitate exploration and exploitation. Such choice of sampling distributions have shown to be successful for black box optimization in large search spaces [29]. First, $\mathcal{P}$ includes a random uniform distribution ($\mathcal{U}$)[7]. In the absence of prior knowledge, our uniform sampling allows for the most effective exploration of the objective $F(\cdot)$. Second, $\mathcal{P}$ contains an adaptively generated sampling distribution, $\mathcal{G}$. This distribution draws on previously observed good samples $\mathcal{S}_g$ with a high EV to locate candidate regions of the search space that can contain a local or global optimum. Specifically, the sampling distribution consists of specialized sampling kernels ($\mathcal{G}$) around good samples, with the effect that new sampling occurs nearby. Assuming mild regularity conditions on the objective function $F(\cdot)$, focusing sampling locally around good samples will likely lead to additional high-quality samples. The goal of the specialized sampling kernels $\mathcal{G}$ is to facilitate search of such neighborhoods.

[7]We use log-uniform sampling when the range of values in the search space is extremely large (e.g., template-variables with dynamic range of $[0, 10^{27}]$).

Figure 4 presents a conceptual example of sampling distributions $\mathcal{P}$ in a 2-dimensional space. The black triangles represent the good samples ($\mathcal{S}_g$) observed in the previous iteration of the sampling algorithm (iteration $i-1$). Learning from the good samples, we generate the specialized sampling distributions ($\mathcal{G}$) in their vicinity, here shown in light blue. A new batch of samples $\mathcal{S}_i$ (shown with dark blue dots) is then sampled using a combination of the specialized kernels $\mathcal{G}$ and the random uniform distribution $\mathcal{U}$ (samples shown with green diamonds) that together compose $\mathcal{P}$.

In the first iteration of optimization, the sampling distribution is initialized to entirely uniform to identify promising regions of the search space. As the optimization progresses, the uniform sampling budget is reduced to 20% and the remaining samples are generated using the specialized exploitation kernels $\mathcal{G}$. In what follows, we explain how the specialized sampling kernels $\mathcal{G}$ are constructed.

5.1.1 *Transaction Reordering.*

A validator can reorder user transactions along with their injected transactions to maximize their EV. Some well-known attacks that can be realized in this way include sandwiching [45] where the validator strategically places certain transactions before and after a user transaction to extract MEV. An important and challenging design consideration in the Lanturn optimizer is complying with the underlying constraints of the blockchain. For this purpose, we devise a novel technique for exploring the space of valid transaction orderings. For a given set of transactions, recall that a reordering is valid *if and only if* it preserves the original order of the transactions of a user.

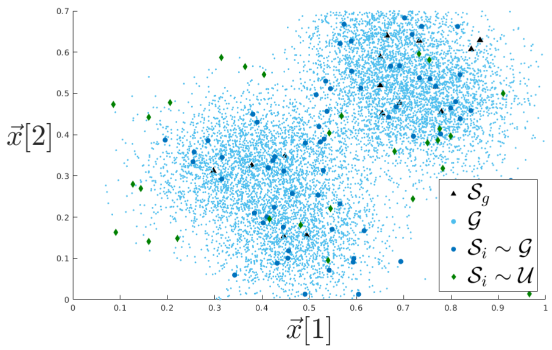


Figure 4: Lanturn adaptively adjusts the underlying sampling distributions by learning from previously evaluated samples with highest EV ($\mathcal{S}_g$). Our samples ($\mathcal{S}_i$) are generated (notation $\sim$) using a combination of specialized kernels ($\mathcal{G}$) and uniform random distribution ($\mathcal{U}$) to balance exploitation and exploration.

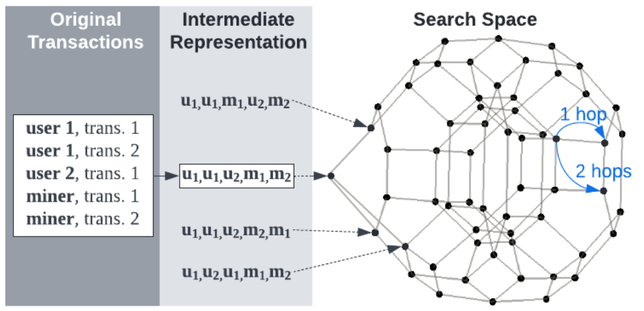


Figure 5: An example transaction block with the corresponding intermediate representation. The optimization search space for transaction ordering is constructed based on the intermediate representation: nodes correspond to unique permutations and edges connect (neighbor) permutations which differ only in one adjacent pair of elements. Lanturn's optimizer learns to traverse this graph and find the transaction ordering that achieves close to the maximal EV.

To directly incorporate this constraint in the Lanturn optimizer, we define our search space over an intermediate representation of a given transaction set such that there is a one-to-one mapping between valid transaction orderings and elements of the intermediate representation. The intermediate representation is obtained by replacing each transaction with its corresponding unique user id as shown in Figure 5. The result is a multiset that can be mapped to a valid ordering of transactions by instantiating transactions for unique user ids according to their nonces.

Our search space is constructed as a graph with nodes corresponding to all unique permutations of the intermediate representation. We define neighbors as permutations where exactly two adjacent elements are swapped, and connect them via an edge in the search space as illustrated in Figure 5. The *intuition* behind this particular way of organizing the search space of reorderings is that swapping adjacent transactions smoothly imitates MEV activity such as frontrunning and backrunning.

We develop a novel optimization algorithm based on adaptive sampling to search the graph and locate the node which yields the maximal EV. From the adaptive sampling perspective, the nodes in the search space correspond to valid samples and the sampling distributions are designed specifically for random graph traversal. In devising our method we were inspired by genetic algorithms [36], which are powerful metaheuristics for black-box optimization. As evident from the name, genetic algorithms aim to mimic the biological process of evolution by encouraging generation of near-optimal samples and eliminating inferior ones with a low objective function value. To achieve this goal, genetic algorithms rely on fitness-proportionate *selection* and random *mutation* to enable local search in the vicinity of previously observed good samples.

Our adaptive sampling relies on similar heuristics to update the sampling distributions and guide the search towards the MEV optima (line 11 in Algorithm 1). Specifically, we create our sampling distribution $\mathcal{G}$ via the selection and mutation operations in genetic algorithms as follows. Once the good samples are identified (line 8 in Algorithm 1), we perform a random selection from $\mathcal{S}_g$, where the probability of selecting a sample is proportional to the MEV it obtained. This operation resembles the fitness-proportionate selection in genetic algorithms where samples with higher scores have a higher chance of being selected.

We define random mutation as performing x random hops from the initial node (sample) over edges in the search space graph. Each hop is simulated by swapping two adjacent elements in the transaction order. We adaptively tune the number of adjacent swaps x as a function of the MEV obtained by the original sample: for higher-MEV samples, we select tighter bounds for local search by reducing x, while for lower-MEV samples, we increase our search radius using a higher x. We linearly map the original MEV range, say $[a, b]$, observed among the selected samples to a probability (p) onto the range $[c, d]$ (specified as a hyperparameter to the learning algorithm) in the inverse order. That is, a sample with EV=a is mapped to probability $p = d$ and a sample with EV=b is mapped to probability $p = c$ where $c < d$, with intermediate values in $[a, b]$ mapped accordingly. Then for each sample, we obtain x by multiplying the corresponding probability p with the length of the sample vector, i.e., number of transactions. Controlling the number of random hops (adjacent transaction order swaps) using the probability p enables us to adaptively tune the boundaries of our search, thus balancing exploration and exploitation.

5.1.2 Transaction Template-Variables. The validator can inject template symbolic transactions where the template-variables have a very large search space. Lanturn automatically learns the best values for these variables such that the validator's EV is maximized. Recall that to this end, Lanturn relies on custom sampling kernels that effectively search the space of valid values in order to locate the EV optima. Our probability distribution functions for adaptive sampling ($\mathcal{G}$) adjust the sampling density across the search space based on previously observed samples. To provide more detail, we construct $\mathcal{G}$ using a Gaussian Mixture Model (GMM), where the parameters are dynamically determined to maximize the likelihood of prior good samples $\mathcal{S}_g$. Prior work in sampling theory [25, 26, 43] proves that Gaussian kernels with adaptive covariances can reconstruct arbitrary non-linear functions. We therefore spatially adjust the covariance of our GMMs based on the location of prior samples in $\mathcal{S}_g$. To comprehensively capture the behavior of the objective function in the vicinity of $\mathcal{S}_g$, our samples are drawn from two GMM components following the methodology proposed in [29].

The first GMM consists of multivariate Gaussian kernels $\mathcal{N}(\cdot, \cdot)$ with their means located on previously observed good samples:

$$\mathcal{G}_1(\vec{x}) \sim \sum_{k=1}^{|\mathcal{S}_g|} \phi_k \cdot \mathcal{N}(\vec{x}_k^g, \Sigma), \tag{3}$$

where ϕ_k denotes the weights assigned to different Gaussian kernels and is proportional to the MEV obtained by the k-th good sample $\vec{x}_k^g$. The covariance Σ is a diagonal matrix where the nonzero elements are determined based on the current range of variables $\vec{x}[i]|_{i=1}^d$ observed among good samples:

$$\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_d), \quad \sigma_i = \frac{1}{3} \cdot \max_{\vec{x} \in \mathcal{S}_g}(\vec{x}[i]) - \min_{\vec{x} \in \mathcal{S}_g}(\vec{x}[i]). \tag{4}$$

The second GMM is constructed using Gaussian kernels placed at the midpoints between pairs of good samples as shown in equation 5. The intuition behind this sampling kernel is to explore the region enclosed between two (local) optima, as their connecting line is likely to be aligned with a gradient ascent trajectory. $\mathcal{G}_2$ is constructed as follows:

$$\mathcal{G}_2(\vec{x}) = \sum_{k=1}^{K} \mathcal{N}(\mu_k, \Sigma_k). \tag{5}$$

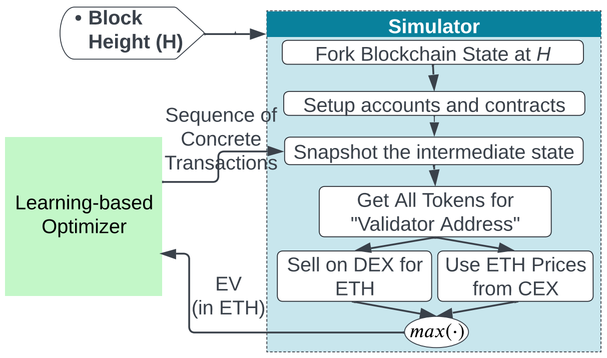
Here, K is the total number of good sample pairs. For each Gaussian kernel above, the mean and covariance matrices are constructed using two good samples $\vec{x}_{k,1}^g$ and $\vec{x}_{k,2}^g$ as follows:

$$\begin{aligned} \mu_k &= \frac{1}{2}(\vec{x}_{k,1}^g + \vec{x}_{k,2}^g), \\ \Sigma_k &= \mathrm{diag}(\sigma_1^k, \ldots, \sigma_d^k), \quad \sigma_i^k = \frac{1}{6} \cdot |\vec{x}_{k,1}^g[i] - \vec{x}_{k,2}^g[i]|. \end{aligned} \tag{6}$$

Aside from the sampling budget assigned to uniform random sampling as explained in Section 5.1, we equally distribute the remaining budget between the two GMMs explained above.

5.2 Simulation

Recall that the optimization module learns from the feedback provided by the simulation module. Figure 6 represents the abstract steps in our simulation module. The simulation module is responsible for executing a sequence of concrete transactions for any given block number, and returning the value accrued to the validator. Our experiments in this work are geared towards the Ethereum blockchain which has the most active and mature DeFi ecosystem. To be able to simulate transactions for any given block number, we utilize an Ethereum *archive* node which stores the complete state after execution of every block in the blockchain. Transactions are simulated on a *local* node instantiated by any Ethereum smart contract development tool (e.g., Hardhat [24] and Foundry [32]). The local node works on a private fork of the blockchain state at a given block height by fetching it from the archive node. We modify



• Block Height (H)
Simulator
Fork Blockchain State at H
Setup accounts and contracts
Snapshot the intermediate state
Get All Tokens for "Validator Address"
Sell on DEX for ETH
Use ETH Prices from CEX
max(·)
Sequence of Concrete Transactions
Learning-based Optimizer
EV (in ETH)

Figure 6: Abstract steps of Simulator

this state to give the validator a certain initial capital (details in Section 6) in the native currency ETH. In order to simulate a given ordered sequence of transactions, we send these transactions in the particular order to the local node and mine a block in the “fifo” order. This configuration disregards the default ordering of mined transactions in the Ethereum protocol according to their gas fees, and instead mines them in the order in which they are received. Note that the development tool allows us to simulate the next block on demand, without waiting for any time period for block production as in the Ethereum protocol. Once the next block is mined, we query the state from the local node to obtain the validator’s balance in ETH. The value accrued to the validator is simply the final balance less the initial capital given to the validator.

Better feedback to the optimization module. The value accrued to the validator could be in ETH and/or any other currency which could be exchanged for ETH. While the feedback given to the optimization module could be only the ETH balance, in line with our definition of the objective function (Section 3.1), we find that a feedback based on the total value accrued to the validator—including both the ETH balance as well as that of any other currency / token—leads to faster convergence of our optimization algorithm. For this purpose, we therefore compute a single scalar representing the equivalent total value denominated in ETH. While computing the best equivalent value of other currencies in terms of ETH is itself a non-trivial problem, a reasonable estimate, as described below, suffices for our purposes. We derive the value of each non-ETH asset x in terms of ETH by either swapping it for ETH on the DEX (UniswapV2/UniswapV3/Sushiswap) which offers the best price for the x-ETH pool, or using prices from a centralized exchange for the x-ETH pair (assuming no slippage on the centralized exchange). We then add this equivalent value of each non-ETH asset to the final ETH balance to get the total value accrued to the validator.

Improving transaction simulation speed. To enhance the speed of performing numerous simulations with different orderings of transactions, we optimize the simulation workflow using the following techniques. Because the Ethereum blockchain state is large in size, the local node only fetches entries in the historical state whenever needed and caches them to minimize the number of queries to the archive node. Actions such as the initialization of the validator’s capital, token approvals, any contract deployment etc.

are identical across different simulations so we keep a snapshot of the state after this step. For each simulation, we reset the state of the local node to the snapshot. Additionally, operations by the local node that are irrelevant to the Ethereum Virtual Machine should be disabled. For example, we don't need to verify the signature or user nonce in a transaction because only valid transactions are fed as input. There's no need to append the mined block to the local chain either because it doesn't affect the resulting EV.

6 EVALUATION

While the techniques of our work are general, our evaluation focuses specifically on Ethereum as it houses the largest and most mature DeFi ecosystem today. Our aim is to experimentally answer the following questions:

(1) How effective is Lanturn at measuring the cryptoeconomic security of smart contracts in the context of real-world user activity? Specifically, for a validator, how much MEV is uncovered by Lanturn at any given block height?
(2) What are the patterns and strategies that Lanturn learns? Can it not only discover existing well-known strategies but also discover new ones?
(3) How does Lanturn perform relative to the computation and time budgets available to it?

We conduct extensive experiments with Lanturn on the historical data from the Ethereum blockchain for three popular AMMs: UniswapV2, Sushiswap and UniswapV3 and one of the most popular lending protocol: AaveV2. Note that we use the same hyperparameters across all our experiments, i.e. Lanturn's learning algorithm is not tuned to a specific smart contract.

Setup. All our experiments are run on an AMD Ryzen Threadripper 3960X with 48 CPU threads, 128 GB RAM and SSD storage. We use the Erigon client as our archive node for serving the blockchain state at any historical block height. For our local node, we utilize the Hardhat software and run 44 parallel instances of it (utilizing all but 4 CPU threads on our server). The optimization module exploits this parallelism to simultaneously simulate multiple concrete transaction sequences that are sampled in the same iteration.

In the remainder of this section, we first describe our dataset (Section 6.1). We then report the MEV extracted by Lanturn (Section 6.2) for each contract both in isolation and when composed. We then analyze the strategies discovered by Lanturn (Section 6.3), report the effectiveness of Lanturn as a function of time and computation budget available (Section 6.4) and finally discuss some implementation considerations(Section 6.5).

Our implementation along with scripts needed to reproduce experiments can be found at https://github.com/lanturn-defi/lanturn. The repository also contains the link to download our dataset.

6.1 Dataset Collection

Flashbots baseline. For a perspective on the MEV extracted by Lanturn, we first establish a baseline for MEV extracted in the wild. It is very challenging to determine an accurate number for the total MEV extracted by all the players in the wild, as MEV activity can look indistinguishable from benign activity. However, thanks to Flashbots, we can arrive at a reasonably tight *lower bound.* Recall that Flashbots hosts MEV auctions where any strategic agent can bid for MEV opportunities by submitting *bundles* of transactions. By May 2021, 84% of the Ethereum mining power was plugged into the Flashbots ecosystem [11], and the MEV auctions had become extremely competitive, with bots routinely bidding away more than 95% of their MEV profits [10]. The transactions in each bundle, along with their bids, are available via the Flashbots API: https://blocks.flashbots.net. We treat these bids as a reasonable tight lower bound for MEV extracted in the wild. For AMM experiments, we obtain our baseline of MEV extracted in the wild by summing up the bids for all the bundles that are a subset of the transactions we feed into Lanturn. For the lending protocol experiment, we obtain our baseline of MEV by summing up the bids for all the bundles that have at least one transaction performing a liquidation on AaveV2. Note that after Ethereum's upgrade to the "Merge" hardfork [4] (Block#15,537,351) in Sept 2022, Flashbots shifted to auctioning off entire blocks. This update prevents us from computing the baseline from the bundle-level data for our experiments, and hence we report our experimental results only for blocks before Block#15,537,351.

Data for Lanturn. We first describe our data collection for AMMs. We use our Ethereum archive node to collect data for every swap and liquidity event on UniswapV2, Sushiswap and UniswapV3. In total, we collected $\approx 23 \times 10^6$ number of swaps, and liquidity events. We then identify the interesting blocks to explore for Lanturn. We first filter the blocks which have significant activity for the DeFi contract of interest — trades and liquidity events of over 500 ETH for Sushiswap and UniswapV2, and over 1000 ETH for UniswapV3 (as it has significantly higher volume than the other two AMMs). This criterion gives us a large—yet not too large—dataset (16,942 blocks) of potentially opportunistic blocks for the validator. For our AaveV2 experiments , we collected data for every liquidation event on AaveV2, as liquidations are the primary MEV activity on lending protocols. In total, we collected 27,961 such events. We then filter those events where the collateral is in the native currency (ETH), obtaining a total of 11,889 events across 7,323 blocks.

We finally filter out the blocks for which we have the aforementioned baseline data. At the end, we obtain a total of 8,128 blocks. For Centralized Exchange (CEX) prices of cryptocurrencies, we utilize a free API from Binance to obtain minute-level historical prices for all USD-based markets. We use the price at the tick nearest to the given block number's timestamp in our simulation module.

6.2 MEV uncovered by Lanturn

We conduct several experiments with AMMs and the lending protocol at different block heights to understand the effectiveness of Lanturn in uncovering MEV and associated strategies in various contexts. We give the validator a starting balance of 1000 ETH (or WETH[8]), unless specified otherwise. Table 1 summarizes our various experiments and documents the strategies (re)discovered by Lanturn. In exploring the MEV extracted from interaction with AMMs, we divide our experiments into two categories: the first category where Lanturn only inserts transactions that interact with a single AMM, and the second category of experiments where Lanturn is allowed to insert transactions that interact with multiple

[8]WETH or Wrapped Ether is an ERC20 complaint form of ETH, and allows for more flexible interactions with smart contracts

Experiment	Strategies
Single AMM - Trading	Frontrunning, Sandwiching, Lazarus, Gas-leeching
Single AMM - Trading and Providing liquidity	All of the above, Frontrunning + Just-In-Time (JIT) Liquidity
Composition of AMMs	All of the above, Arbitrage
Lending Protocol	Backrunning Price Oracle

Table 1: Summary of experiments and strategies discovered by Lanturn. Strategies highlighted in green are previously undocumented(Section 6.3). '+' denotes combination of the two strategies.

AMMs. We also compare the results from these two categories to empirically show the effect of composition on the cryptoeconomic security of smart contracts.

Input to Lanturn. Recall that Lanturn requires three inputs: the blockchain state, the user transactions available for reordering, and the templates for validator insertions along with any constraints on the template-variables. A given block B specifies the blockchain state to Lanturn. While we could theoretically feed all the transactions appearing in B to Lanturn, most transactions have no effect on the specific smart contracts of our interest. Therefore, the transactions provided to Lanturn are a subset of the transactions appearing in B, thereby reducing the search space of reorderings. To obtain this subset of transactions, we start with the set of transactions in B that interact with the particular smart contract of interest. We then repeatedly expand this set (until the point it no longer grows) by adding transactions from B which interact with the accounts that also interact with the transactions already present in the set. This is done so that we include all the transactions that directly or indirectly affect the smart contracts of interest[9]. The pseudocode for this procedure is given in Appendix B. Note that the transactions input to Lanturn can contain transactions from MEV bots or other transactions privately communicated to the validator, both of which can be exploited by Lanturn in our experiments. *While our Flashbots baseline does not capture this exploitation, it still serves as a useful lower bound for giving a perspective on the MEV uncovered by Lanturn.* We specify the template symbolic transactions along with each experiment below.

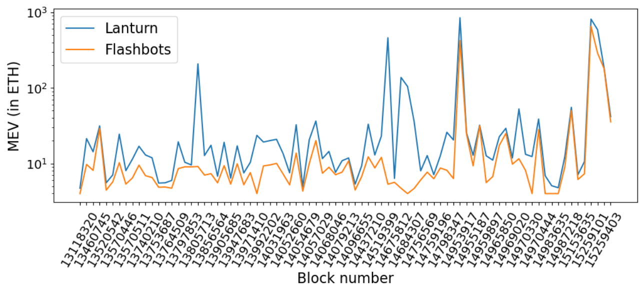


Figure 7: Per block MEV extracted by Lanturn versus the flashbots baseline on UniswapV2.

[9]Reordering of transactions affects their dynamic execution and consequently the accounts with which they interact. Our method is a middle ground between covering all the possibly relevant transactions and including irrelevant transactions which cause a blowup in the search space

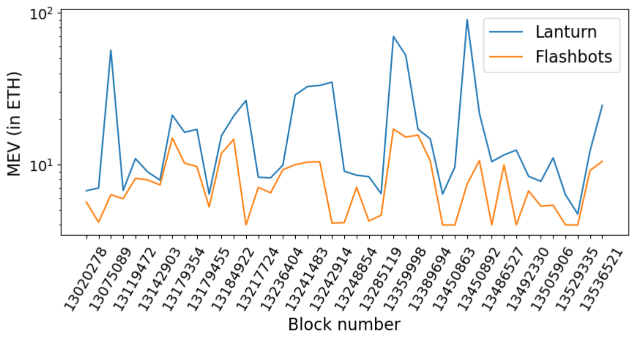


Figure 8: Per block MEV extracted by Lanturn versus the flashbots baseline on Sushiswap.

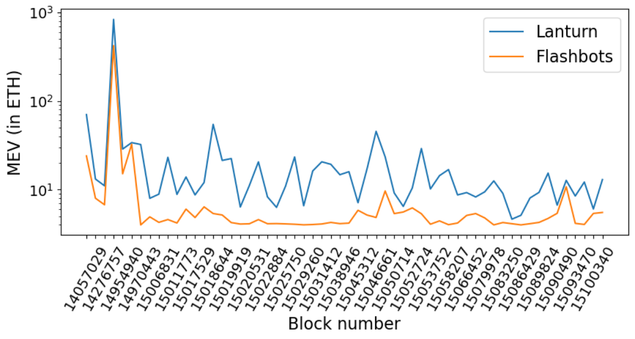


Figure 9: Per block MEV extracted by Lanturn versus the flashbots baseline on UniswapV3.

Trading on a single AMM. We evaluate Lanturn for a validator making trades on each AMM in isolation. Figures 7, 8 and 9 show the MEV uncovered by Lanturn while trading with Sushiswap, UniswapV2 and UniswapV3 respectively. Lanturn consistently uncovers higher MEV than the baseline for MEV extracted on each AMM. This is due to Lanturn learning the popular sandwiching [45] and frontrunning [37] strategies as well as some new strategies that we discuss in Section 6.3. For each pool (market for exchanging, say, `token0` and `token1`) appearing in the user transactions, Lanturn makes use of two template insertions—thus allowing the validator to swap in either direction. Figure 10 shows one of these templates for both UniswapV2 and UniswapV3 (Sushiswap smart contracts are similar to UniswapV2). The template-variable α_1 represent the

UniswapV2 Swap

"from" : Validator, "to" : UniswapV2Router(0x7a250...2488D), "value": α_1, "function":swapExactETHForTokens, "calldata": {"amountOutMin":0, "path":[`token0`,`token1`], "to":Validator, "deadline":INT_MAX}

UniswapV3 Swap:

"from" : Validator, "to" : UniswapV3Router(0xE5924...61564), "value": 0, "function":exactInputSingle, "calldata": {"tokenIn":`token0`, "tokenOut": `token1`, "fee": `fee`, "recipient": Validator, "deadline": INT_MAX, "amountIn": α_1, "amountOutMinimum":0, "sqrtPriceLimitX96":0}

$$0 \leq \alpha_1 : \mathtt{int} \leq 1000 \times 10^{18}$$

UniswapV3 Provide Liquidity:

"from" : Validator, "to" : PositionManager, "value": 0, "function":addLiquidity, "calldata": {"tokenIn":`token0`, "tokenOut": `token1`, "fee": `fee`, "liquidity": α_2, "tickLower": α_3, "tickUpper": α_4, }

$$0 \leq \alpha_2 : \mathtt{int} \leq 10^{25}$$
$$-887272 \leq \alpha_3, \alpha_4 : \mathtt{int} \leq 887272$$

AaveV2 LiquidationCall:

"from" : Validator, "to" : AaveV2LendingPool, "value": 0, "function":liquidationCall, "calldata": {"collateralAsset": `ETH`, "debtAsset": `token0`, "user": `user`, "debtToCover": α_5, "receiveAToken": `false` }

$$0 \leq \alpha_5 : \mathtt{int} \leq 10^{25}$$

Figure 10: Template transaction insertion for each AMM

amount of `token0` traded by the validator in exchange for a quantity of `token1`. The domain of α_1 reflects the initial capital available to the validator (measured in wei, 1 ETH = 10^{18} wei).

Trading and providing liquidity on a single AMM. UniswapV3 allows a Liquidity Provider (LP) to control the price range in which she is providing her liquidity, in contrast to first generation AMMs (like UniswapV2), which fixed the price range to $(0, \infty)$. This policy allows an LP to more effectively target her liquidity, but at the same time requires an LP to be more strategic as her profits crucially depend on her choice of range [15]. We conduct experiments that evaluate Lanturn when the validator is not only able to trade but also provide liquidity on UniswapV3. Because liquidity provisioning is a capital-intensive action, we grant the validator a higher working capital of 10,000 ETH. For UniswapV3, Figure 13 shows the additional MEV uncovered by Lanturn through liquidity provisioning and trading on top of the MEV uncovered only through trading. The positive value indicates additional MEV which results from Lanturn learning a lesser-known arbitrage strategy called "Just-In-Time (JIT) Liquidity". Here, the validator injects large amounts of liquidity just before users' trades, and withdraws the capital immediately in the same block. In the process, the validator is able to earn a disproportionate share (compared to other LPs) of the fees paid by the users. Recall that LP's profits are determined not only by the revenue earned from the fees but also the loss incurred due to price deviations. We observe that Lanturn learns a sophisticated strategy: it combines multiple users' trades (in opposite directions) such that

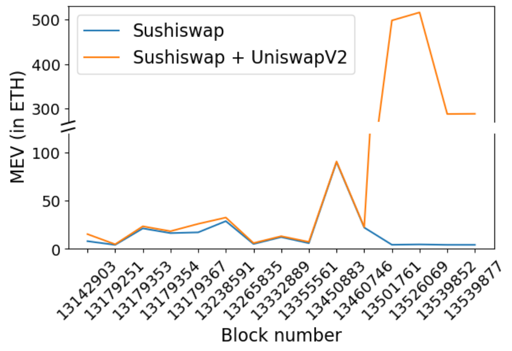


Figure 11: Per block MEV extracted by Lanturn on Sushiswap versus combination of Sushiswap and UniswapV2.

the price deviation for the validator (and thus the loss) is minimized. We also observe that Lanturn combines the JIT strategy with sandwiching and frontrunning strategies. Figure 10 shows the template insertion for providing liquidity. Template insertion for removing liquidity is similar, and shares all the three template-variables of the liquidity provision transaction. Template-variable α_2 influences the liquidity provided (or removed) by the validator, whereas α_3 and α_4 determine the price range in which the validator's liquidity is active. The domain of α_2 covers the initial capital available with the validator in our experiments, while the domain of α_3 and α_4 is the widest possible range allowed by UniswapV3.

Composition of AMMs. We additionally evaluate Lanturn on *composed* smart contracts. Lanturn in this case explores a space of actions by the validator that may include transactions affecting any of the composed contracts of interest. This is achieved by giving Lanturn the user transactions interacting with any of the composed contracts and also allowing Lanturn to insert transactions based on templates of all the composed contracts. Figure 11 shows the MEV uncovered by Lanturn when Sushiswap is composed with UniswapV2 as opposed to only interacting with Sushiswap. Similarly, Figure 12 shows the MEV uncovered by Lanturn when UniswapV2 is composed with UniswapV3 as opposed to only interacting with UniswapV2. Clearly, the composition of smart contracts unlocks huge MEV opportunities. We see that Lanturn, without encoding the specific strategies, learns to exploit arbitrage opportunities across AMMs, acquire assets from the cheapest AMM, provide and remove liquidity strategically across AMMs, etc.

Lending Protocol. We evaluate Lanturn on the composition of lending protocol AaveV2 and AMMs to demonstrate that Lanturn can generalize beyond AMMs. Recall that the primary MEV activity on lending protocols involves performing a liquidation, which involves paying the debt (usually denominated in stablecoins) of an undercollateralized position and obtaining the underlying collateral (usually denominated in ETH) at a discount. We compose AaveV2 with Sushiswap and UniswapV3 so that the validator can acquire the stablecoins or other non-native tokens from the AMMs and use them to pay off the debt on AaveV2 during a liquidation call. Because liquidations are capital intensive, we grant the validator a higher initial capital of 10,000 ETH. Figure 10 shows the liquidation template for Aave2, wherein the template-variable α_5 represents

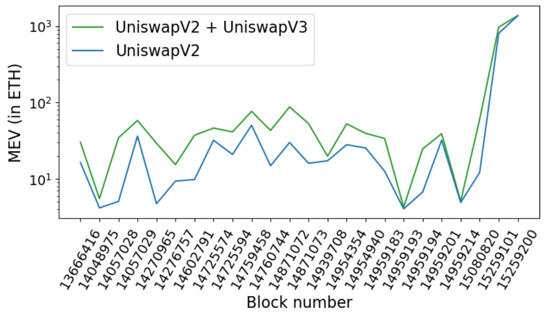


Figure 12: Per block MEV extracted by Lanturn on UniswapV2 versus combination of UniswapV2 and UniswapV3.

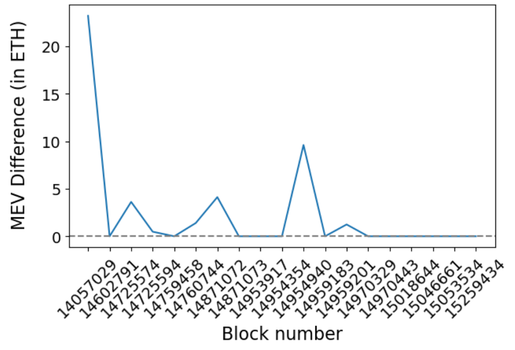


Figure 13: UniswapV3: Additional MEV uncovered by Lanturn through liquidity provisioning and trading on top of the MEV uncovered only through trading. A positive value indicates profits due to the JIT-liquidity strategy, while a zero value indicates no additional profits from liquidity provisioning.

the debt paid off by the validator. Finally, in order to focus our results on MEV activity on AaveV2 and not include purely AMM activity such as arbitrage, we only include the AMM templates that swap in one direction: swaps from ETH to stablecoins/non-native tokens. We remove any user transactions that perform swaps or provide/remove liquidity on the AMMs, so that the validator does not engage in frontrunning or sandwiching.

Figure 14 shows the MEV uncovered by Lanturn for the composition of AaveV2 with Sushiswap and UniswapV3. Lanturn is able to consistently uncover higher MEV than the baseline. We observe that Lanturn learns the popular backrunning strategy for lending protocols: a transaction that updates the price oracle and causes a debt position to not be sufficiently collateralized is backrun with the inserted liquidation transaction from the validator. The validator, as a result, profits by obtaining the liquidated collateral at a discount.

6.3 Discovering new MEV Strategies

Besides re-discovering known strategies, Lanturn also discovers *novel, previously undocumented* strategies for extracting MEV. An agent can even hard-code these strategies (and emergent heuristics) discovered during the offline exploration of Lanturn into strategy-specific bots that can be run in real-time much more efficiently.

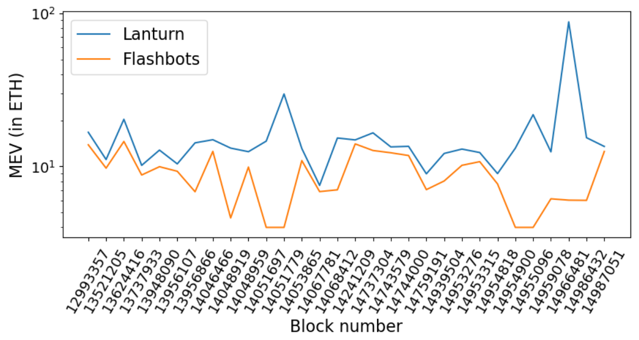


Figure 14: Per block MEV extracted by Lanturn versus the flashbots baseline, evaluated on a combination of Sushiswap, UniswapV3, and AaveV2 transactions.

Lazarus[10] Strategy. User transactions interacting with a DEX usually throw an error ("revert") when they do not get their minimum desired price. These transactions appear in a block as reverted transactions, and include payment of fees to the validator. For example, one transaction that surfaced in our experiments, 0x367e86...08cc33, was sent to the mempool[11] by an MEV bot (0x793FF6...cD24E1). This transaction interacts with UniswapV3 pools, but appears as reverted (in block 14954940) because the bot did not get the desired price. On discovering such failed transactions, Lanturn reorders them to a different position in the block so that the transaction succeeds (brings the transaction "back to life"), but it then also frontruns the very same transaction to extract value for the validator.

Gas-leeching. Gas-leeching is a previously undocumented strategy in which the validator exploits a lack of safeguards in some bots' smart contract to receive significantly high transaction fees from the bot. Recall that bots extract MEV primarily by submitting ordered sequences ("bundles") of transactions to validators via an auction mechanism hosted by relayers such as Flashbots. It is known that "unbundling" bots' transactions opens up the bots to being a victim of frontrunning and sandwiching attacks themselves by the validator. Note that the trust in Flashbots and validators allows bots to take this risk. However, in a notable recent attack [6], the validator "unbundled" bots' transactions and executed sandwich attacks on them to extract more than $20 million.

We find that, contrary to popular belief, frontrunning and sandwiching is NOT the only strategy available to the validator after unbundling the bots' transactions. Lanturn uncovers a new strategy called Gas-leeching, which we now explain through an example.

One such instance of Gas-leeching uncovered during our experiments exploits a popular sandwiching bot 0x000000...416B40. When the transactions submitted by this bot are reordered, the transactions themselves run into an infinite loop and end up consuming all the available gas. Combine this behaviour with the fact that the bot routinely sets an unusually high gas price (as the MEV auction is based on the effective gas price [9]), and the result is the bot

[10]Reference to the New Testament figure Lazarus, who is restored to life after death. Lanturn figures out a strategy that brings a dead transaction back to life.
[11]It is not uncommon for MEV bots to send transactions to the mempool, instead of sending it the host of the MEV auction, usually for flying under the radar of different players involved the MEV auction

paying the validator significant transaction fees while also leaving the original opportunity for the validator to exploit. This example also highlights the weakness of the trust-based MEV auctions.

6.4 Performance

Lanturn can be adapted to execute within a specific budget of computation and/or wall-clock time. To demonstrate the effect of limiting resources devoted to Lanturn, we measure the progress over time of MEV discovery (as a percentage of the final value Lanturn converges to) for each of the blocks in our experiments . Figure 15 shows this progress in Lanturn for different quartiles of blocks. We see that half of the Lanturn executions uncover more than 95% of their MEV in less than 100 seconds. While we run 44 parallel simulation instances for our experiments on a 24-core (48 CPU threads) server, we observe a near linear speedup in our performance as we increase the number of cores available to Lanturn on our server (a speedup of 2X as we increase the cores from 5 to 10, and a further speedup of 1.7X as we increase the cores to 20). The performance of Lanturn can be further increased with higher parallelism, thanks to the many independent samples in each iteration of Lanturn's adaptive learning algorithm.

We now briefly outline how Lanturn can be used in real-time for MEV extraction. In all our experiments above, we simulated the samples of template-variables (44 such samples in one iteration) from the inner loop concurrently, but executed the reordering samples (at least 10 in one iteration) from the outer loop sequentially. This indicates a straightforward 10X improvement in performance with 10 servers, each of which independently evaluates one reordering sample. Observe that this gain amounts to uncovering 95% of our final MEV in less than 10 seconds on a majority of the blocks. A latency of 10 seconds is sufficient to bid in MEV auctions in real-time on Ethereum today [3], which has an inter-block interval of 12 seconds. Note that it is straightforward for Lanturn to also include any newly arriving transactions on the fly. Additionally, while we utilized a stable albeit slower javascript-based simulation library, there are significant performance gains to be unlocked from utilizing a lower latency backend such as the Rust-based REVM[12], which is under active development.

Efficiency of Lanturn as templates grow. The performance of our implementation is bottlenecked by the simulation of the concrete transactions (simulation module), while the time taken in sampling by the optimization module is negligible. Hence, the marginal impact of increasing the number of template transactions on efficiency is dictated by the number of template transactions as a fraction of the total number of transactions. In a majority of our experiments, user transactions are more than 40 and the validator transactions based on templates are between 2 to 9, and hence, the performance degradation from adding more templates is graceful. The template-variables are sampled independently and efficiently. Hence increasing their number mimimally impacts efficiency.

6.5 Implementation Considerations

Interplay among dimensions. It is not uncommon for multiple template-variables in a transaction to have mutual dependencies.

[12]https://github.com/bluealloy/revm

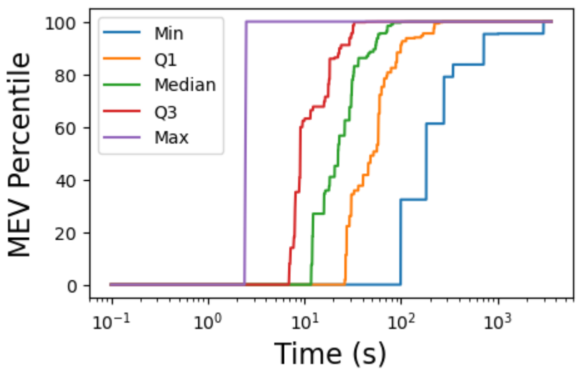


Figure 15: Convergence time of Lanturn adaptive learning to different percentiles of the maximum MEV. Each line shows the different quartiles across the optimizations runs for our experiments run using 44 CPU threads. Lines on the left represent the segments of the problems that converge faster than the segments represented by lines on the right.

For example, consider the liquidity addition transaction in Figure 10. A valid transaction must have $\alpha_3 < \alpha_4$ for a valid price range. While we do not encode these contract-specific relationships, we see that Lanturn quickly learns relationships among the template-variables thanks to the adaptive distribution kernels $\mathcal{G}$ (Section 5.1) in Lanturn's optimizer. However, the initial exploration of Lanturn can be made more efficient if the template-variables can be rewritten into independent dimensions. For example, α_4 can be rewritten as $\alpha_3 + \alpha_4'$, where α_4' denotes the width of the range. The new template-variables α_3 and α_4' can now be explored efficiently again by independent sampling.

Automating template specification. Templates for a smart contract can in principle be derived automatically by inspecting the signature of the public non-view functions in the smart contract interface. However, certain parameter types such as "`bytes`" and "`address`" in some method signatures cannot be meaningfully sampled automatically from the complete domain, without introducing a heuristic, such as sampling addresses from a set of known recent addresses. Further research is required to automate the template generation in a general way. While the need to design templates manually is currently a limitation of Lanturn, templates need to be designed only once for a contract and can be shared (perhaps by the contract developers themselves) among all practitioners of Lanturn across the ecosystem.

Limitations due to learning. Like any learning algorithm, Lanturn requires at least mild regularity conditions in the search space. Smart contracts are Turing complete programs, and can, in theory, be crafted to have arbitrary irregular surface for cryptoeconomic security in terms of the variables of optimization. Fortunately, however, smart contracts for the DeFi applications used in practice have relatively continuous, or even smooth surfaces with respect to MEV exposure, i.e., cryptoeconomic security, due to their underlying algorithmic nature. For example, AMMs owe their smooth surface to the underlying bonding curve, such as the constant product curve in Uniswap.

7 RELATED WORK

Qin et al. [35] investigated the concept of optimizing a blockchain attacker's profits through constrained optimization. The optimization was performed using the Sequential Least Squares Programming (SLSQP) algorithm from the SciPy library. The focus of this work was on optimizing a particular known ordered sequence of transactions already carried out by an attacker on the Ethereum blockchain. The objective function was manually derived for this sequence of transactions in terms of the transaction parameters.

Prior works such as [44] optimize the attacker's profits efficiently for a specific class of arbitrage strategy based on currency-price differentials. The solution is to find the most profitable cycle comprising of currency swaps in DEX markets.

Bartoletti et al. [17] have given an efficient optimal procedure for maximizing profits on constant-function market makers (without concentrated liquidity). The model in this work is purely theoretical and does not consider practical aspects of blockchain execution such as gas costs, which play a crucial role in arriving at profitable strategies that can be executed on the blockchain.

There is a plethora of work [27] applying learning techniques to traditional finance. However, the main goal in this line of literature is to predict the price movements from market structure and external signals. Smart contracts have a crucial difference: Their execution is deterministic and transparent, thus allowing a strategic agent to precisely control and determine future execution results.

8 CONCLUSION

In this work, we have introduced Lanturn—a general purpose, learning-based framework to measure the cryptoeconomic security of smart contracts. Lanturn works in tandem with the native blockchain environment, and its outputs can be executed on the blockchain as is, without modification. Thanks to our formulation of the cryptoeconomic-security problem as an optimization problem with an accompanying adaptive learning algorithm, Lanturn is scalable with respect to both problem size and the complexity of the smart contracts involved.

We have shown experimentally that Lanturn consistently discovers significant MEV compared to a baseline derived from the value extracted by strategic agents in the wild. Lanturn can also not only discover existing strategies, but entirely new ones that are previously undocumented. Lanturn empowers developers to understand the cryptoeconomic behavior of their smart contracts in isolation, as well as when composed with other applications. Blockchain researchers can utilize Lanturn to better understand the incentives of consensus participants, developers can use Lanturn to design MEV-minimizing contracts, and users can benefit from Lanturn by learning (and perhaps reducing) the extractable value exposed in their transactions.

ACKNOWLEDGMENTS

We thank James Austgen, Andrés Fábrega, Smruti Manjunath and anonymous ACM CCS'23 reviewers for their feedback on drafts of this work. This work was funded by NSF TILOS AI institute award number CCF-2112665, NSF grants CNS-2112751, CNS-1564102, CNS-1704615, and CNS-1933655 as well as generous support from the Uniswap Foundation and IC3 industry partners. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and may not reflect those of these sponsors.

REFERENCES

[1] Central limit order book. https://en.wikipedia.org/wiki/Central_limit_order_book. Accessed: 2023-05-04.
[2] Defillama. https://defillama.com/. Accessed: 2023-05-04.
[3] Ethereum block value analytics. https://payload.de/data/. Accessed: 2023-08-01.
[4] The merge. https://ethereum.org/en/roadmap/merge/. Accessed: 2023-05-04.
[5] Nomad bridge hack: Root cause analysis. https://medium.com/nomad-xyz-blog/nomad-bridge-hack-root-cause-analysis-875ad2e5aacd. Accessed: 2023-05-04.
[6] Post mortem: April 3rd, 2023 MEV-Boost relay incident and related timing issue. https://collective.flashbots.net/t/post-mortem-april-3rd-2023-mev-boost-relay-incident-and-related-timing-issue/1540. Accessed: 2023-05-04.
[7] Compound finance, Retrieved 2023. https://compound.finance/.
[8] dydx, Retrieved 2023. https://dydx.exchange/.
[9] Flashbots, Retrieved 2023. https://www.flashbots.net/.
[10] Flashbots MEV leaderboard, Retrieved 2023. https://explore.flashbots.net/leaderboard.
[11] Flashbots transparency report — may & june 2021, Retrieved 2023. https://medium.com/flashbots/flashbots-transparency-report-may-2021-8aff6ff97e9f.
[12] Aave. Protocol whitepaper 2.0, 2020. https://github.com/aave/protocol-v2/blob/master/aave-v2-whitepaper.pdf.
[13] Hayden Adams. Uniswap docs, 2023. https://uniswap.org/docs.
[14] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core, 2021. https://uniswap.org/whitepaper-v3.pdf.
[15] Andreas A Aigner and Gurvinder Dhaliwal. Uniswap: Impermanent loss and risk profile of a liquidity provider. *arXiv preprint arXiv:2106.14404*, 2021.
[16] Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. Clockwork finance: Automated analysis of economic security in smart contracts. In *IEEE S&P*, 2023.
[17] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch Lafuente. Maximizing extractable value from automated market makers. In Ittay Eyal and Juan Garay, editors, *Financial Cryptography and Data Security*, pages 3–19, Cham, 2022. Springer International Publishing.
[18] Lorenz Breidenbach, Christian Cachin, Benedict Chan, Alex Coventry, Steve Ellis, Ari Juels, Farinaz Koushanfar, Andrew Miller, Brendan Magauran, Daniel Moroz, et al. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. *Chainlink Labs*, 1, 2021.
[19] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C Myers. Compositional security for reentrant applications. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1249–1267. IEEE, 2021.
[20] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash Boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *IEEE S&P*, pages 910–927, 2020.
[21] Ding-Zhu Du and Panos M Pardalos. *Minimax and applications*, volume 4. Springer Science & Business Media, 1995.
[22] Michael Egorov. Stableswap - efficient mechanism for stablecoin, 2019. https://classic.curve.fi/files/stableswap-paper.pdf.
[23] Yu Feng, Emina Torlak, and Rastislav Bodik. Precise attack synthesis for smart contracts. *arXiv preprint arXiv:1902.06067*, 2019.
[24] Nomic Foundation. Hardhat: Ethereum development environment for professionals. https://hardhat.org/.
[25] Keaton Hamm. Nonuniform sampling and recovery of bandlimited functions in higher dimensions. *Journal of Mathematical Analysis and Applications*, 450(2):1459–1478, 2017.
[26] Thomas Hangelbroek and Amos Ron. Nonlinear approximation using gaussian kernels. *Journal of Functional Analysis*, 259(1):203–219, 2010.
[27] Bruno Miranda Henrique, Vinicius Amorim Sobreiro, and Herbert Kimura. Literature review: Machine learning techniques applied to financial market prediction. *Expert Systems with Applications*, 124:226–251, 2019.
[28] Yoichi Hirai. Defining the Ethereum virtual machine for interactive theorem provers. In *FC*, pages 520–535, 2017.
[29] Mojan Javaheripi, Mohammad Samragh, Tara Javidi, and Farinaz Koushanfar. Adans: Adaptive non-uniform sampling for automated design of compact dnns. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):750–764, 2020.
[30] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 65–68, 2018.
[31] MakerDAO. The maker protocol: MakerDAO's multi-collateral dai (mcd) system, 2020. https://makerdao.com/en/whitepaper/.
[32] Foundry Organization. Foundry book. https://book.getfoundry.sh/.
[33] Daejun Park, Yi Zhang, and Grigore Rosu. End-to-end formal verification of Ethereum 2.0 deposit smart contract. In *CAV*, pages 151–164, 2020.

[34] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *IEEE S&P*, pages 198–214, 2022.
[35] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. Attacking the DeFi ecosystem with flash loans for fun and profit. In *FC*, pages 3–31, 2021.
[36] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.
[37] 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 *30th USENIX Security Symposium (USENIX Security 21)*, pages 1343–1359, 2021.
[38] Ron van der Meyden. On the specification and verification of atomic swap smart contracts. In *ICBC*, pages 176–179, 2019.
[39] Gottfried Vossen. Database transaction models. In *Computer Science Today*, pages 560–574. 1995.
[40] Xin Wan and Austin Adams. Just-in-time liquidity on the Uniswap protocol. https://blog.uniswap.org/jit-liquidity. Accessed: 2023-05-04.
[41] Ben Weintraub, Christof Ferreira Torres, Cristina Nita-Rotaru, and Radu State. A flash(bot) in the pan: Measuring maximal extractable value in private pools. In *IMC*, page 458–471, 2022.
[42] Gavin Wood. Ethereum yellow paper, 2014. https://github.com/ethereum/yellowpaper.
[43] Yiming Ying and Ding-Xuan Zhou. Learnability of gaussians with flexible variances. *Journal of Machine Learning Research*, 8(Feb):249–276, 2007.
[44] Liyi Zhou, Kaihua Qin, Antoine Cully, Benjamin Livshits, and Arthur Gervais. On the just-in-time discovery of profit-generating transactions in DeFi protocols. In *IEEE S&P*, pages 919–936, 2021.
[45] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. High-frequency trading on decentralized on-chain exchanges. In *IEEE S&P*, pages 428–445, 2021.

A DECENTRALIZED FINANCE, CONSTANT FUNCTION MARKET MAKERS AND LENDING PROTOCOLS

DeFi, is a term used to denote the ecosystem of financial smart-contracts and protocols on (permissionless) blockchains (such as Ethereum). DeFi protocols have already been deployed for a wide range of use cases, and allow users to borrow, lend, exchange, or trade assets on a blockchain. An especially popular class of DeFi protocols are *decentralized exchangees* (DEXes), which enable users to swap cryptocurrency tokens without requiring a trusted intermediary. While a DEX can be designed in several ways, the most successful design is called an Automated Market Maker (AMM). We provide a brief background on the workings of AMMs, and specially focus on the most popular AMMs – Sushiswap, UniswapV2 and UniswapV3, which collectively hold more than $4.5 billion in their smart contracts as of April 2023.

AMM. An AMM consists of different markets (called “pools”) for facilitating exchange between (usually a pair of) cryptocurrencies (called “tokens”). For each pool, Liquidity Providers (LPs) provide liquidity to the AMM pool by depositing the involved cryptocurrencies in a certain proportion into the underlying smart contract. Unlike Central Limit Order Books (CLOBs [1]) in traditional exchanges, AMMs do not require LPs to specify a fixed price for which their assets can be offered to a trader. Rather, AMMs allow for passive liquidity provision – the underlying smart contract determines the *appropriate* price at which the liquidity is made available to a trader, and credits the fees from the trader to the LP. We now discuss relevant details of this price mechanism and liquidity provision.

UniswapV2 and Sushiswap. The token exchange rate for a liquidity pool in popular AMMs such as UniswapV2 and Sushiswap is determined programmatically by a *constant-product curve* [13]. Specifically, if the pool contains x tokens of X and y tokens of Y (x and y are called pool’s “reserves”), then across all trades the invariant $xy = c$ will hold for some constant c. For instance, when a trader sells Δx tokens of X to the contract, she will receive Δy tokens of Y such that $(x+\Delta x)(y-\Delta y) = c$. The price for exchanging an infinitesimal amount of X for Y is $\frac{y}{x}$, and can range from $(0, \infty)$. Liquidity providers can add tokens to the pool for users to swap against and earn fees based on their fraction of the pool’s total liquidity. LPs incur a loss (called “impermanent loss” [15]) if the price moves away from the price at which they locked in liquidity into the contract. The profits of an LP are thus determined by these losses from price deviations and revenue from the earned fees. For details, we refer the reader to [15].

UniswapV3. An updated version—UniswapV3—was published in May 2021 and currently is the dominant DEX, with monthly volumes as high as $50 billion. UniswapV3, while maintaining the constant product curve as its core mechanism, mainly changes the operation of liquidity provisioning. Instead of requiring liquidity to be provided in the $(0, \infty)$ range, UniswapV3 allows for *concentrated liquidity*—i.e., liquidity that is *active* only if the price is inside the LP’s chosen interval $[p_a, p_b]$. This results in a “shifted” price curve given by $(x + \sqrt{c/p_b}) \cdot (y + \sqrt{c/p_a}) = c$. For a more detailed description, we refer the reader to [13, 14]. The main consequence of interest to our work is that the fees earned by the LP now depend directly on her chosen price interval.

Lending Protocol: AaveV2. AaveV2 [12] is a liquidity protocol which allows lenders to pool their assets through a smart contract and allows borrowers to borrow these assets after locking sufficient collateral in the smart contract. A portion of the interest paid by the borrowers accrues to the lenders through the interest rate mechanism [12]. A borrower’s position becomes available for liquidation when the value of the locked collateral falls below a certain threshold relative to the value of its debt and interest charges. Once a position is available for liquidation, anyone can repossess the collateral by repaying the position’s debt at a discounted rate. In order to determine the value of collateral and debt assets, the protocol utilizes an external price feed from a price oracle.

B FILTERING USER TRANSACTIONS

Algorithm 2 Identifying user transactions for input to Lanturn

```
Inputs: Block $B$, Contracts $C$
Outputs: User transactions $T$
1:  $T = \emptyset$
2:  $\mathcal{A} = \emptyset$                                  ▷ Interacting addresses
3:  $\mathcal{S}$ = Transactions in $B$ whose execution trace intersections
    with $C$                                             ▷ Seed transactions
4:  while $\mathcal{A}$ grows in size do
5:      $\mathcal{A}.add$(set of sender and destination addresses of
         transactions in the set $\mathcal{S}$)
6:      for $tx \in B$.transactions do
7:          if $tx$.sender $\in \mathcal{A}$ or $tx$.destination $\in \mathcal{A}$ then
8:              $\mathcal{S}.add(tx)$
9:  for $tx \in B$.transactions do
10:     if $tx$.sender $\in \mathcal{A}$ or $tx$.destination $\in \mathcal{A}$ then
11:         $T.add(tx)$
12: return $T$
```

We now give the pseudocode for our procedure to filter user transactions input to Lanturn. The aim of this procedure is to identify all the transactions from a given block that directly or indirectly affect the state of the given smart contracts. Due to the dynamic nature of execution of blockchain transactions, it is not possible to statically determine all the transactions that affect the state of the smart contracts in any of the possible reorderings that Lanturn may explore. Therefore, our procedure is designed to strike a balance between identifying all the possibly relevant transactions and yet, not include too many transactions that have no bearing on the state of the smart contracts.

Algorithm 2 shows the pseudocode for our procedure.