Nyx: Detecting Exploitable Front-Running Vulnerabilities in Smart Contracts

Wuqi Zhang[†], Zhuo Zhang[‡], Qingkai Shi[‡], Lu Liu[†], Lili Wei[¶], Yepang Liu[‖], Xiangyu Zhang[‡], Shing-Chi Cheung[†*]

[†] *The Hong Kong University of Science and Technology, {wuqi.zhang, lliubf}@connect.ust.hk, scc@cse.ust.hk*
[‡] *Purdue University, {zhan3299, shi553}@purdue.edu, xyzhang@cs.purdue.edu*
[¶] *McGill University, lili.wei@mcgill.ca*
[‖] *Southern University of Science and Technology, liuyp1@sustech.edu.cn*

2024 IEEE Symposium on Security and Privacy (SP) | 979-8-3503-3130-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/SP54263.2024.00146

***Abstract*—Smart contracts are susceptible to front-running attacks, in which malicious users leverage prior knowledge of upcoming transactions to execute attack transactions in advance and benefit their own portfolios. Existing contract analysis techniques raise a number of false positives and false negatives in that they simplistically treat data races in a contract as front-running vulnerabilities and can only analyze contracts in isolation. In this work, we formalize the definition of exploitable front-running vulnerabilities based on previous empirical studies on historical attacks, and present *Nyx*, a novel static analyzer to detect them. *Nyx* features a Datalog-based preprocessing procedure that efficiently and soundly prunes a large part of the search space, followed by a symbolic validation engine that precisely locates vulnerabilities with an SMT solver. We evaluate *Nyx* using a large dataset that comprises 513 real-world front-running attacks in smart contracts. Compared to six state-of-the-art techniques, *Nyx* surpasses them by 32.64%-90.19% in terms of recall and 2.89%-70.89% in terms of precision. *Nyx* has also identified four zero-days in real-world smart contracts.**

1. Introduction

Smart contracts, designed for operation on decentralized blockchain platforms like Ethereum, form a vast interconnected network that drives the innovative financial system known as Decentralized Finance (DeFi). Despite its groundbreaking potential, DeFi, much like traditional finance, is susceptible to certain vulnerabilities. One such vulnerability carried over from traditional finance in DeFi is *"front-running"*, an illegal practice where unethical brokers leverage knowledge of upcoming transactions to make trades that benefit their own portfolios. For instance, a broker, upon receiving an order to buy a large number of stocks, could foresee the subsequent price surge and purchase some shares in advance. Once the original order is processed and the stock price rises, the broker can then sell his shares to realize a profit. In the DeFi ecosystem, this vulnerability is even more prominent. As all transactions, essentially smart contract function invocations, are publicly pending in a transaction pool before they are executed on the blockchain [22], anyone has access to this information and can potentially attack it.

While traditional finance discourages front-running by legally penalizing such actions, whereas relies on the smart contracts themselves to deter and prevent this. These contracts should be designed in a manner that there is no profit incentive for front-running, thus mitigating the propensity of this unethical behavior. Maximal Extractable Value (MEV) [17] is used by the Ethereum community as a metric to quantify the total amount of profits in all conceivable front-running attacks. A recent study has highlighted that these front-running attacks on Ethereum have led to substantial financial damages, with losses surpassing 675 million USD before the Merge update of Ethereum (September 2022) [7]. As of July 2023, profits from front-running have exceeded the sum of 224K ethers since the Merge [25].

The substantial impact of front-running attacks within the blockchain community is well acknowledged, yet the existing techniques fall short in effectively identifying vulnerabilities that render systems susceptible to these attacks. These vulnerabilities often stem from flawed designs, which allow an adversary to make profits by front-running a user's transaction. A significant obstacle in the detection of such vulnerabilities lies in the design of the detection oracle.

Existing studies have taken significant efforts to pinpoint specific code behaviors susceptible to front-running, such as transaction order dependency [31], event ordering bugs [30], and state inconsistency bugs [14], which are essentially data races between transactions [42]. However, while these methods are effective for detecting targeted behaviors, they have trouble assessing the profitability or *"exploitability"* of a potential front-running attack, rendering a large number of false positives. For instance, a data race between transactions could be an intentional design feature and may not necessarily offer a profitable

* *Correspoinding author.*
The work was done during Wuqi Zhang's visit at Purdue University.
Yepang Liu is with both the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems at Southern University of Science and Technology.

DOI 10.1109/SP54263.2024.00146

exploitation avenue for front-running attacks. Research by Perez and Livshits [36] reveal that only 54 out of 1,881 contracts (about 2.9%) flagged by existing tools were ever exploited via front-running attacks. In our evaluation, we find that existing tools raise, on average, 51.12% false alarms in a benchmark of real-world contracts [54].

The recent work by Zhang et al. [54] provides insightful observations on historical front-running attacks, highlighting that rational attackers exploit data races in smart contracts only when there is a potential profit by front-running transactions. Leveraging this observation, we formalize the definition of front-running vulnerabilities within the context of smart contract analysis, with particular emphasis on the exploitability of data races across multiple contracts (Section 2). Specifically, we assert that a DeFi application, characterized as a group of interconnected smart contracts, contains front-running vulnerability if and only if there exist two competing execution paths such that the execution of the first one adversely affects the subsequent one, leading to financial loss. The two execution paths correspond to the attack and victim transactions, respectively. The negative influence one poses on the other typifies the essence of a front-running attack.

Existing techniques, however, encounter substantial limitations in terms of scalability to facilitate cross-contract analysis to detect front-running vulnerabilities that can be exploited. Existing contract analysis techniques usually focus on a single execution path, often overlooking the mutual influence between execution paths executed by two transactions. Detecting the existence of such vulnerability requires the modeling of all combinations of two execution paths and evaluating the influence that one poses on the other under different execution orders. Therefore, our formalized front-running vulnerability definition cannot be directly applied as a detection oracle with existing techniques. In addition, the business logic of many DeFi applications is implemented with multiple interconnected contracts. Given that existing techniques only analyze contracts in isolation, they fail to adequately comprehend and analyze the complete logic of a DeFi application, thus overlooking many vulnerabilities. In our evaluation, we find that existing tools miss a majority of the vulnerabilities in a benchmark of real-world contracts [54].

To tackle the limitations of existing works, we present *Nyx*, an innovative technique for front-running vulnerability detection. *Nyx* is designed to efficiently prune benign or possibly intentional instances of data races and identify exploitable front-running vulnerabilities effectively. Particularly, when analyzing a DeFi application, *Nyx* examines all possible pairs of contract functions, where one function is invoked by an attacker and the other is invoked by a victim. For each function pair, *Nyx* aims to assess the profit changes resulting from altering the invocation order and raises an alert if these profit shifts align with the previously stated definition of exploitability.

The major challenge stems from the vast search space, given that *Nyx* needs to explore all paths that victims and attackers may execute across multiple contracts. To mitigate this, *Nyx* first incorporates an innovative and efficient static pruning technique. We propose a necessary condition for the vulnerability that allows us to soundly prune the functions deemed non-vulnerable from the search space. We propose the extended system dependency graph (xSDG) to capture the control and data flow across multiple contracts in a DeFi application. Furthermore, an efficient algorithm is designed to verify the necessary condition leveraging graph reachability analysis using Datalog. Our experiment results show that static pruning is capable of pruning 88.73% non-vulnerable function pairs rapidly.

The retained function pairs after pruning are then accurately analyzed by a tailored symbolic execution engine. *Nyx* symbolically models profit changes (resulting from altering the invocation order of the attacker and the victim) and utilizes the capabilities of an SMT solver to determine if the profit changes meet the criteria set by the vulnerability oracle. Our evaluation indicates that *Nyx* can achieve 90.19% recall and 70.89% precision on a large-scale benchmark consisting of 513 real-world attacks in smart contracts [54]. Our contributions are summarized as follows.

- We devise a novel static pruning technique that effectively identifies profit-related data races among functions in a given DeFi application and prunes a considerable portion of the search space.
- We present a tailored symbolic execution engine capable of modeling profit changes and leveraging the power of SMT solver to validate whether the profit changes satisfy the criteria established by the exploitability definition.
- We develop a prototype of *Nyx* and assess its efficacy using a large-scale front-running attack benchmark [54]. In comparison with six state-of-the-art front-running vulnerability detection techniques, *Nyx* outperforms them by 32.64%-90.19% in terms of recall and 2.89%-70.89% in precision. Regarding runtime overhead, *Nyx* aligns with existing solutions. Notably, *Nyx* successfully identifies four zero-days in real-world DeFi projects. *Nyx* is publicly available at https://github.com/Troublor/Nyx.

Threat Model. We aim to detect vulnerabilities that can be exploited by front-running attacks, wherein adversaries manipulate the transaction execution order to maximize their profits at the expense of other users. We hence consider vulnerabilities that necessitate adversaries to supply specially crafted inputs, such as providing an exceedingly large value to induce an integer overflow, out of the scope. Additionally, vulnerabilities related to the manipulation of block properties, such as block timestamp manipulation attacks [2], or those that exploit weak pseudo-random number generators [13], are beyond the scope of this paper. These types of vulnerabilities, although important, fall outside the scope of this paper.

2. Front-Running Vulnerability

The previous empirical studies characterize and mine historical front-running attacks on the blockchain by identifying the profits that the attacker can make and the loss the

victim suffers [46], [54]. In this section, we formally define the front-running vulnerability in smart contracts from the aspect of program analysis based on the observations of these previous studies.

Notations. Blockchain can be considered as a state transition system, where each state σ is characterized by the data stored on the blockchain. One state transits to another when a transaction is executed. A blockchain state σ maps contract storage variables to concrete values. The *storage variables* of a contract are global variables whose values are shared by different transactions calling the contract. Let $\mathcal{F}$ be a list of all functions in a DeFi application under analysis, represented as a group of interconnected contracts $\mathcal{C}$. We define a transaction as a tuple $T = \langle a, f(\vec{x})\rangle$, where a is the transaction submitter, $f \in \mathcal{F}$ is the first function invoked by T, and $\vec{x}$ is the arguments passed to f in the transaction T. We denote a state transition of transaction T as $\sigma \xrightarrow{T} \sigma'$, where σ and σ' are the blockchain states before and after executing T, respectively. Transactions on the blockchain are executed sequentially. We use T_1T_2 to denote executing T_1 sequentially before T_2. The state transition of executing T_1T_2 is denoted as as $\sigma \xrightarrow{T_1T_2} \sigma_{T_1T_2}$.

Assets. An asset is a valuable entity that attackers may target as potential gains and victims may lose in an attack. For instance, the assets can be the amount of tokens (e.g., ethers, ERC20 tokens, NFTs) that a user possesses. The definition of assets can be contract-specific according to the functionalities of a DeFi application. The exploitability of front-running vulnerability is defined in terms of the gain and loss of assets for the attacker and the victim. In the subsequent discussion, let $\mathcal{A}_\sigma(a)$ denote the assets that can be possessed by a transaction submitter, a, at a blockchain state σ.

In the vulnerability detection technique to be proposed in Section 4, the definition of assets can be customized as per the logic of specific DeFi applications so that the oracle can be applied to the detection of vulnerabilities under specific use cases. By default, we implement the assets as standard tokens a user may possess, including ether [52], ERC20 [50], ERC721 [20], ERC777 [16], and ERC1155 [41] tokens. Considering the historical front-running attacks collected by previous studies [46], [22], [54], implementing all these standard tokens as assets is sufficient to capture most of the vulnerabilities.

Definition 1 (Front-Running Vulnerability)**.** *We say a pair of functions $\langle f_1, f_2\rangle$, where $f_1, f_2 \in \mathcal{F}$, is vulnerable to front-running attack if and only if $\exists T_1, T_2, \sigma, a_1 \neq a_2$, where*

$$T_1 = \langle a_1, f_1(\vec{x_1})\rangle,\ \ T_2 = \langle a_2, f_2(\vec{x_2})\rangle,$$
$$\sigma \xrightarrow{T_1T_2} \sigma_{T_1T_2},\ \ \sigma \xrightarrow{T_2T_1} \sigma_{T_2T_1},$$

such that the following condition is satisfiable:

$$\mathcal{A}_{\sigma_{T_1T_2}}(a_1) > \mathcal{A}_{\sigma_{T_2T_1}}(a_1)\ \ \wedge\ \ \mathcal{A}_{\sigma_{T_1T_2}}(a_2) < \mathcal{A}_{\sigma_{T_2T_1}}(a_2).$$

Note that f_1 and f_2 can be different functions, i.e., an attacker may call a different function to launch a front-running attack on the victim. The definition of front-running vulnerability characterizes the incentive of an attacker to launch a front-running attack and the damage that the attack makes to a victim user. The predicate $\mathcal{A}_{\sigma_{T_1T_2}}(a_1) > \mathcal{A}_{\sigma_{T_2T_1}}(a_1)$ asserts that the attacker a_1 gains profits (i.e., the assets of a_1 is more) if the attacker's transaction T_1 is executed before the victim's transaction T_2 (front-running). Similarly, the predicate $\mathcal{A}_{\sigma_{T_1T_2}}(a_2) < \mathcal{A}_{\sigma_{T_2T_1}}(a_2)$ asserts that the victim a_2 suffers from loss (i.e., the assets of a_2 is less) if T_2 is executed after T_1. The predicate $a_1 \neq a_2$ makes sure that the attacker and the victim are different users.

Discussion. The condition in Definition 1 captures the mutual influence between the attacker and victim. Other scenarios, where attackers make profits but no victim is harmed, are not considered vulnerable by Definition 1. A significant majority of front-running vulnerabilities involve direct victims, as confirmed by measurement studies [46], [39] and our analysis of 119 audit reports from a leading contract audit platform, Code4rena [3]. Similarly, scenarios, where attackers are not affected by the victim transactions, are also excluded in Definition 1 since attackers cannot make profits in front-running.

Definition 1 is capable of capturing vulnerabilities that require attackers to submit multiple transactions to exploit. One example is sandwich attack [57], where an attacker submits two attack transactions before and after the victim transaction, respectively, to manipulate token exchange prices and realize price manipulation profits. However, pinpointing vulnerabilities inducing such attacks does not require synthesizing the entire exploits with multiple attack transactions. Those multi-transaction attacks still exhibit profitable transaction order dependencies between the victim and one of the attack transactions. The vulnerabilities thereby can be identified by Definition 1. As evidence, we conduct experiments of *Nyx* on a dataset consisting of a large number of sandwich attacks (Appendix E), showing that *Nyx* can achieve a recall as high as 95.9%.

The root cause of the vulnerability is the *profitable transaction order dependency* between two function invocations. Transaction order dependencies are common in smart contracts but only those leading to profitable opportunities for attackers are considered as harmful and a true vulnerability. Definition 1 checks such profitability to avoid reporting those benign ones. Developers avoid front-running vulnerabilities by either eliminating transaction order dependencies or preventing attackers from making profits if such dependencies are unavoidable in the business logic.

3. Motivation

In this section, we illustrate the front-running vulnerability with a real-world DeFi application as an example. Then, we discuss the limitations of existing vulnerability detection techniques, followed by a sketch of our solution.

3.1. Example Contracts

Fig. 1 shows three contracts of a DeFi application that allow project owners to raise crowdfund for their ongoing projects (Contracts `CrowdFund` and `Controller`). Each

```
1  contract CrowdFund {
2   Controller controller;
3   mapping(address => uint) projects;
4   mapping(uint => ERC20) tokens;
5   mapping(uint => uint) donations;
6   function donate(uint project, uint donation)
        public {
7    tokens[project].transferFrom(msg.sender, this,
         donation);
8    donations[project] += donation;
9   }
10  function withdraw() public {
11   uint project = projects[msg.sender];
12   uint donation = donations[project];
13   tokens[project].transfer(msg.sender,donation);
14   donations[project] = 0;
15  }
16  function _changeToken(address projOwner, ERC20
        newToken) public {
17   require(msg.sender == controller);
18   require(newToken.owner()==address(this));
19   require(noProject(newToken));
20   uint project = projects[projOwner];
21   ERC20 oldToken = tokens[project];
22   tokens[project] = newToken;
23   oldToken.transferOwnership(projOwner);
24  }}
25 contract Controller {
26  CrowdFund crowdFund;
27  function changeToken(ERC20 newToken)public{
28   require(notBlackList(msg.sender));
29   ERC20 oldToken = crowdFund._changeToken(msg.
         sender, newToken);
30  }}
31 contract ERC20 {
32  address public owner;
33  function transferOwnership(address newOwner)
        public onlyOwner {
34   owner = newOwner;
35  }}
```

Figure 1: Example smart contracts adapted from real-world contracts [23], [37]. Function pair ⟨changeToken, changeToken⟩ is vulnerable [37].

project is associated with an ERC20 token contract as the type of crowdfund to collect (the mapping variable tokens at Line 4). The CrowdFund contract contains a donate function that enables donors to donate tokens to a specific project (Lines 6–8), and a withdraw function that enables project owners to withdraw the donations for their own project (Lines 13–14). Project owners can change the token contract associated with their own project using the function changeToken in contract Controller (Lines 27–29). The ownership of the old token contract associated with the project will be transferred to the project owner.

The contracts contain a front-running vulnerability related to the function pair ⟨changeToken, changeToken⟩ in contract Controller. To associate a token contract with a project, a project owner (victim) needs to first transfer the ownership of newToken to contract CrowdFund (due to the precondition at Line 18) and then call function changeToken. The project owner's invocation to changeToken, denoted by $T_2 = \langle victim, \texttt{changeToken(newToken)} \rangle$, can be attacked by a malicious user who calls changeToken, denoted as $T_1 = \langle attacker, \texttt{changeToken(newToken)} \rangle$, before the project owner and associates newToken, which should have belonged to the victim, to the attacker's project. Then, the project owner's invocation T_2 will fail at Line 19 since the newToken has already been associated with the attacker. As a result, the attacker steals the ownership of newToken from the project owner (victim). Fitting into Definition 1, the asset can be defined as the total number of ERC20 contracts that the victim or attacker owns. When executing T_1T_2, i.e., attacker's transaction before the victim's one, the newToken will be associated to attacker's project (Line 22), instead of to the victim's project. Note that one user can easily claim the ownership of a token contract associated with their own project. After the execution, the attacker steals the ownership of $newToken$ from the victim as profits, i.e., $\mathcal{A}_{\sigma_{T_1T_2}}(attacker) = \mathcal{A}_{\sigma}(attacker) + 1$, $\mathcal{A}_{\sigma_{T_1T_2}}(victim) = \mathcal{A}_{\sigma}(victim)$, where σ is the blockchain state before T_1 and T_2 are executed. Similarly, when executing T_2T_1, $\mathcal{A}_{\sigma_{T_2T_1}}(victim) = \mathcal{A}_{\sigma}(victim) + 1$, $\mathcal{A}_{\sigma_{T_2T_1}}(attacker) = \mathcal{A}_{\sigma}(attacker)$. Thus, The condition defined in Definition 1 is satisfied.

To fix the vulnerability, contract Controller may perform additional identity verification such that only the caller, who transfers the ownership of newToken to CrowdFund, can invoke changeToken. If authentication is enforced, the contract will no longer be considered vulnerable by Definition 1 since the $a_1 \neq a_2$ condition is violated (i.e., both attacker and victim must be the authenticated user).

3.2. Limitations of Existing Techniques

We leverage our example in Fig. 1 to illustrate the two typical limitations of existing techniques.

Cross-Contract Analysis. They cannot detect the vulnerability in the example contracts since they only analyze contracts individually and do not analyze the logic across multiple contracts. Front-running vulnerabilities usually lie in the designed logic of multiple contracts of a DeFi application [54]. In Fig. 1, analyzing contract CrowdFund alone cannot detect the vulnerability since the function can only be called by the trusted address, i.e., the Controller contract (Line 17). The contract Controller itself is also safe since although the function can be called by anyone, there are no critical operations. In addition, the profits targeted by attackers in the front-running are often managed by external contracts, e.g., in Fig. 1, transfers of the ERC20 tokens (Line 13) or the contract ownership (Line 23) are performed in external contracts. Analyzing individual contracts cannot analyze the complete logic and may miss many vulnerabilities. According to our experiments in Section 5.1, existing tools miss over 90% vulnerabilities due to this limitation.

Exploitability. The detection oracle of existing techniques is inappropriate for catching vulnerabilities, inducing false alarms. Existing techniques detect vulnerabilities by identifying possible data races between two transactions invoking the same contract. However, many contract data races are unexploitable since they are benign or intended [36]. For instance, there exists a data race between transactions invoking function donate and withdraw on the shared variable

`donations`. However, such a data race cannot be exploited by attackers to earn extra benefits because the donations of a project can only be withdrawn by its project owner. Attackers may change the amount of donations withdrawn by the project owner by invoking `donate` before the victim. However, the amount of donations can only increase, meaning that the victim does not suffer from any loss from the attack. Thus, determining vulnerabilities by means of data races without considering the exploitability can induce many false alarms. Our evaluation finds that existing techniques, on average, generate 54.92% false positives due to this limitation.

3.3. Challenge and Our Solution

In Section 2, we formalize the front-running vulnerability in smart contracts, taking into account the exploitability of data races across multiple contracts and various kinds of profits attackers may target. This, however, requires extensive analysis of various asset transfers across contract variables, which collectively prescribe a large search space, in particular when the analysis includes the detection of vulnerabilities arising from cross-contract interactions. Thus, we design an exploitability-aware and cross-contract-semantic-aware static pruning approach to reduce cross-contract symbolic execution paths that are not exploitable for front-running attackers. Then, symbolic execution is performed on function pairs with deferred path constraint solving and several optimizations to detect vulnerabilities against Definition 1 efficiently. Below, we summarize the key challenge and our solution.

Challenge. Detecting the vulnerability using Definition 1 as oracle is challenging due to the increased complexity of program analysis. To capture possible updates of contract variables and asset transfers, one may need to symbolically execute all pairs of possible functions that can be called by an attacker and a victim, respectively. The executions cover possible transaction execution orders between attacker transactions and victim transactions. An SMT solver can then be used to check if an execution would result in profit for the attacker and losses for the victim. However, this straightforward approach is not scalable because it needs to symbolically execute all possible path combinations of each function pair. The total number of paths that symbolic execution needs to explore is $O(n^2)$, where n is the number of execution paths in a public function. Note that n itself is already exponential to the number of branches in the program. Worse still, external function calls also need to be analyzed during symbolic execution to support analyzing the semantics across multiple contracts, making the number of execution paths even larger. The simple design would lead to severe path explosion [29].

Our Solution. We tackle the challenge by constructing an extended system dependency graph (xSDG) on smart contracts and pruning the search space substantially using static analysis. A tailored symbolic execution engine that can co-analyze two functions (from the same contract or multiple different contracts) is then utilized to identify exploitable front-running vulnerabilities. In the following, we highlight some of the important features of our technique.

Search Space Pruning. To address the challenge, we propose a datalog-based static analysis to effectively prune the search space by eliminating those function pairs whose execution orders cannot violate the vulnerability oracle. This is based on an extended system dependency graph (xSDG) that characterizes control and data flow across multiple contracts, as well as their interactions. For example, the function pair $\langle$`withdraw`, `withdraw`$\rangle$ in Fig. 1 will be pruned by our analysis, since each transaction submitter can only withdraw the donation of the project led by themselves. We can statically conclude that a transaction executing `withdraw` would not induce losses to another transaction submitted by a different user. The function pair is thus excluded from the expensive symbolic execution. The evaluation shows that our static pruning can reduce the search space by 88.73% on the benchmark.

Oracle Checking. The static pruning can confidently preclude function pairs that are not vulnerable. However, there are still false alarms. To faithfully check whether an attacker can launch a front-running attack to obtain profits and cause loss to victims, we adopt SMT solvers to check the satisfiability of constraints encoding our oracle. To obtain the constraints, we construct a symbolic blockchain state and symbolically execute two symbolic transactions, T_1 (attacker) and T_2 (victim), in two orders, i.e., T_1T_2 and T_2T_1, respectively, to simulate the two scenarios that the victim is attacked and the front-running does not occur. In order to mitigate the path explosion problem, we merge the symbolic values of variables at the merge points of different execution paths so that the SMT solver does not need to check the feasibility of each path. For each function pair, we collect symbolic expressions representing the digital assets possessed by the two transaction submitters of T_1 and T_2 after the execution of two orders. Finally, we construct constraints to encode our oracle. The amount that an attacker can profit from and that a victim may lose are deduced by comparing the assets possessed by the attacker and the victim under different execution orders of their transactions. The SMT solver is only invoked once for each function pair to check the satisfiability of constructed constraints, instead of being invoked for each execution path, since we have merged the values of variables in different paths and the SMT solving is deferred until all branches have been explored. Our approach can effectively detect vulnerabilities with 90.19% recall and 70.89% precision in our evaluation. Despite of the drastically increased search space compared to existing tools, our evaluation indicates that the runtime overhead of our approach is comparable to that of existing tools.

4. Methodology

Fig. 2 shows an overview of *Nyx*. *Nyx* takes a smart contract group, which is a set of contracts closely interacting with each other, as input. *Nyx* first extracts the SlithIR [24], an intermediate representation of contracts,

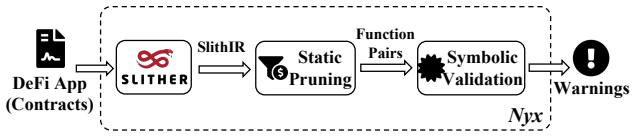


Figure 2: Workflow of *Nyx*

using Slither [47]. Then, *Nyx* conducts a static pruning based on SlithIR to eliminate function pairs that do not satisfy the necessary condition of front-running vulnerability (Section 4.1). In this phase, most function pairs are filtered out. After that, *Nyx* validates whether each suspicious function pair is truly vulnerable to front-running with Definition 1 (Section 4.2) via symbolic execution and SMT solving. In the end, *Nyx* reports all the vulnerable function pairs.

4.1. Static Pruning by Necessary Condition

As mentioned in Section 3.3, a simple symbolic execution would suffer from severe performance issues due to the large search space of checking our vulnerability oracle along each pair of paths (Definition 1). To tackle this, we propose a novel static analysis to first reduce the search space before adopting the symbolic execution approach. Intuitively, *Nyx* checks a necessary condition for a function pair to be vulnerable and eliminates those not satisfying the condition.

Necessary condition (informal). *A pair of functions $\langle f_1, f_2\rangle$ is vulnerable only if executing f_1 influences the profits a user can make in function f_2, and the execution of f_2 also influences the profits made in f_1.* The profits in the execution of a function refer to the change of assets of the user who invokes the function. Intuitively, f_1 influencing the profits in f_2 is the necessary condition for the front-running to be harmful to the victim a_2, i.e., $\mathcal{A}_{\sigma_{T_1T_2}}(a_2) < \mathcal{A}_{\sigma_{T_2T_1}}(a_2)$ in Definition 1. Similarly, f_2 influencing the profits in f_1 is the necessary condition for the front-running to be profitable for the attacker a_1, i.e., $\mathcal{A}_{\sigma_{T_1T_2}}(a_1) > \mathcal{A}_{\sigma_{T_2T_1}}(a_1)$.

Example. Fig. 3a shows a vulnerable contract, which rewards the first caller who provides a solution string whose cryptographic hash is a specific value. The function pair $\langle$`solve`, `solve`$\rangle$ is vulnerable since when a normal user (victim) who finds a correct solution calls function `solve`, an attacker can copy the solution and call `solve` before the victim to steal the rewards that should be given to the victim. Such an attack is possible since normal users' submitted transactions are kept in a public pending pool before execution, and attackers can easily inspect the content of pending transactions (e.g., the solution for the function `solve`). Suppose an attacker a_1 submits a transaction $T_1 = \langle a_1, \texttt{solve}(s_1)\rangle$ which is executed before $T_2 = \langle a_2, \texttt{solve}(s_2)\rangle$ submitted by victim user a_2. When T_1 is executed before T_2, the execution of T_1 updates the storage variable `bounty` and `pubFund` to zero after the `reward` (i.e.,`bounty` + `pubFund`) is sent to a_1 (Line 7). The data written to the storage variable influence the later execution of T_2. The `reward` in T_2 is zero, making the victim a_2 receive no transferred ethers. Similarly, if T_2 is executed before T_1, the user a_2 will receive the reward while a_1 receives nothing. In other words, one execution of the function `solve` updates the storage variable so that the profits made in another invocation of the function become zero.

On the other hand, consider the function pair $\langle$`solve`, `addFund`$\rangle$ in Fig. 3a, where the users a_1 and a_2 send transactions of these two functions, respectively. The execution of function `addFund` may influence the profit that the user a_1 can make in function `solve`, i.e., changing the storage variable `pubFund` which is part of the reward. However, the execution of `solve` cannot influence the profits of user a_2 in function `addFund`. The profit of a_2 is always $-$`msg.value` since a_2 pays a constant amount of ethers to the contract. The execution of `solve` does not make the a_2 pay more ethers than a_2 would want to. In other words, user a_2 does not have losses even if his transaction is executed after that of a_1. Therefore, this function pair is considered not vulnerable, without the need for symbolic execution. □

This static pruning is guided by a graphic program representation, called an extended system dependency graph for smart contracts (Section 4.1.1), and a set of Datalog rules enacting the necessary condition of vulnerability (Section 4.1.2 and Section 4.1.3).

4.1.1. Extended System Dependency Graph (xSDG).

We propose an extended version of the system dependency graph (SDG) [28] on smart contracts to capture the control and data dependency across contracts within a transaction and between transactions. The extended system dependency graph (xSDG) is represented as $G = \langle N, E, \lambda\rangle$, where N is a set of nodes, E is a set of edges, and λ is a function mapping each $e \in E$ to a label. Fig. 3b shows the corresponding xSDG for the code in Fig. 3a.

Nodes. One node in the xSDG G is either a statement or a storage variable, i.e., $N = N_S \cup N_V$. N_S contains all statements (e.g., `msg.sender.transfer(reward())`) as well as the entry node for each function (e.g., entry `solve`) and function formal parameter assignments (e.g., `solution = solution`$_{\texttt{in}}$ with `solution` the formal parameter at Line 5 of Fig. 3). Nodes in N_V represent the locations in the blockchain state where storage variables store, e.g., `bounty` and `pubFund`.

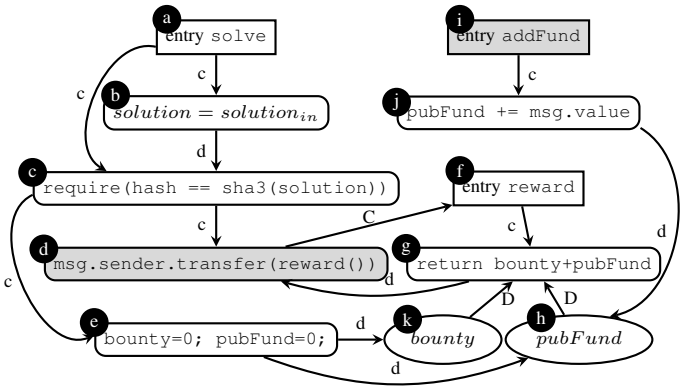
Edges. An edge between two nodes represents either a control dependency or a data dependency relationship. Let E_c denote the control dependency edges, and E_d denote the data dependency edges. Then, $E = E_c \cup E_d$. Function $\lambda : E \to \{\texttt{c}, \texttt{C}, \texttt{d}, \texttt{D}\}$ is a mapping from each edge to a label. For an edge $e = \langle n, u\rangle \in E_c$, $\lambda(e) =$ `c` if node u control-depends on node n[1]; $\lambda(e) =$ `C` if node n calls a function whose entry node is u. For an edge $e = \langle n, u\rangle \in E_d$, i.e., there is a data flow from node n to u, $\lambda(e) =$ `D` if $n \in N_V$, otherwise $\lambda(e) =$ `d`, which will be explained in the following.

Difference from SDG. Our extended system dependency graph (xSDG) differs from the traditional system dependency graph (SDG) [28] in that we distinguish the *intra-transaction* and *inter-transaction* data flow. As mentioned

1. Node u control-depends on n if and only if the execution of u is conditionally guarded by n.

```
1  contract FindThisHash {
2    bytes32 public hash = 0xb5b...e0a;
3    uint bounty = 1 ether;
4    uint pubFund = 0;
5    function solve(string solution) public {
6      require(hash == sha3(solution));
7      msg.sender.transfer(reward());
8      bounty = 0; pubFund = 0;
9    }
10   function reward() view public {
11     return bounty + pubFund;
12   }
13   function addFund() public payable {
14     pubFund += msg.value;
15   }
16 }
```

(a) Contract under analysis



(b) Extended system dependency graph (xSDG) of the contract

Figure 3: Example of extended system dependency graph (xSDG) for a contract code snippet. Edge labels c, C, d, and D in the graph refer to control dependency, inter-procedural call, intra-transaction data dependency, and inter-transaction data dependency, respectively. Colored nodes are profiting nodes.

previously, front-running attacks are essentially data races between transactions. Contract storage variables, such as `bounty` and `pubFund` in Fig. 3a, are the shared data between transactions. Therefore, we refer to the data flow edges from contract storages as **inter-transaction data dependency**, i.e., $\lambda(e) = D$, since they flow from one transaction to the other. Other data flows are referred to as **intra-transaction data dependency**, i.e., $\lambda(e) = d$, since they flow within a single transaction. By distinguishing intra- and inter-transaction data flow, we can interpret whether the change of user assets (e.g., token transfers) can be affected by another transaction via inter-transaction data flow since storage is the data shared across transactions.

Cross-contract. The xSDG captures the control and data dependency within a **contract group** $\mathcal{C}$, which is a set of contracts that interact with each other closely to implement functionality as a whole, e.g., contract `CrowdFund` and `Controller` in Fig. 1. In practice, the contract group is the set of contracts provided by developers to be analyzed by our approach. In the xSDG, we do not distinguish function invocations within a contract or across contracts. However, determining callee in external contracts is hard since the addresses of external contracts are usually not known in static analysis. We adopt an over-approximation approach of inferring cross-contract function callees within the contract group $\mathcal{C}$ by consider all functions in $\mathcal{C}$ with the same signature as potential callees. Although, as mentioned in Section 3.2, the search space grows drastically as more contracts are included in $\mathcal{C}$, our static analysis approach can efficiently prune much of the search space as discussed in what follows.

4.1.2. Necessary Condition of Vulnerability.

In this section, we formally define the necessary condition of a function pair to be vulnerable based on the xSDG.

Definition 2 (Profiting Node)**.** *A profiting node is a node in the xSDG where transaction submitter, a, makes profits, i.e., $\mathcal{A}_\sigma(a)$ changes.*

Note that the profits made in a profiting node may be either positive or negative, depending on whether the assets of the user increase or decrease. We identify profiting nodes by finding statements performing asset transfer actions since transfers are those actions changing users' assets. Such transfer actions are explicitly defined as function or event interfaces by the specification of token standards we support as digital assets. For example, statements invoking a ERC20 `transferFrom` function or emitting ERC20 `Transfer` event are considered as profiting nodes. Specially, we also consider the entrance of a *payable* function as a profiting node since callers can implicitly pay ethers to the underlying contract when invoking such functions.

Example. The profiting nodes in Fig. 3b are ⓓ and ⓘ since the caller of the function `solve` receives the reward at node ⓓ, and the caller of `addFund` pays `msg.value` amount of ethers to the contract at node ⓘ. □

As mentioned previously, given a function pair $\langle f_1, f_2 \rangle$, the necessary condition of the vulnerability requires the profiting nodes executed in one function can be influenced by the execution of the other function. Therefore, for each function, we define the set of profiting nodes that can be executed in the function and the set of profiting nodes that the function can influence through inter-transaction data flow, respectively.

Definition 3 (Reachable Profiting Nodes)**.** *Given an xSDG $G = \langle N, E, \lambda \rangle$, the reachable profiting nodes of f, denoted by $RP(f)$, is the set of profiting nodes reachable on G^* from the entry node of f, where $G^* = \langle N, E^* \rangle$ and $E^* = \{e | e \in E, \lambda(e) \in \{c, C, d\}\}$.*

Definition 4 (Influenced Profiting Nodes)**.** *Given an xSDG $G = \langle N, E, \lambda \rangle$, and let D denote the set of storage variable nodes that are reachable on G^* from the entry node of f, the influenced profiting nodes of f, denoted by $IP(f)$, is the set of profiting nodes reachable on G' from any $n \in D$, where $G' = \langle N, E' \rangle$, where $E' = \{e | e \in E, \lambda(e) \in \{c, d, D\}\}$, and G^* is defined same as in Definition 3.*

Subgraph G^* only contains the control dependency (`c`), function call (`C`), and intra-transaction data flow (`d`) edges,

thus only capturing the nodes that can be reached (via either control flow or data flow) by the execution of a single transaction. Subgraph G' contains the inter-transaction data flow (D) but not the function call (C) edges, thus capturing the control and data-dependency relationship on the storage variables, which may be written by another transaction. $RP(f)$ denotes the set of profiting nodes that can be executed by a transaction invoking f. $IP(f)$ represents the set of profiting nodes that f can influence in the execution of other transactions, i.e., function f may write to storage variables, which profiting nodes of other transactions depend on.

Example. For the function `solve` in Fig. 3, $RP(\texttt{solve}) = \{$ⓓ$\}$ since node ⓓ in function `solve` transfers rewards to the transaction submitter. In addition, $IP(\texttt{solve}) = \{$ⓓ$\}$ since function `solve` updates the storage variable `bounty` and `pubFund`, which later will influence another transaction that executes node ⓓ. □

Lemma 1 (Necessary Condition of Definition 1)**.** *A pair of functions $\langle f_1, f_2\rangle$ is vulnerable to front-running attacks only if $IP(f_1) \cap RP(f_2) \neq \emptyset \wedge IP(f_2) \cap RP(f_1) \neq \emptyset$.*

Proof. Suppose an attacker a_1 is able to submit an attack transaction T_1 invoking f_1 to execute before T_2 invoking f_2 by the victim user a_2. From the Definition 1, the execution of the transactions must satisfy $\mathcal{A}_{\sigma_{T_1T_2}}(a_1) > \mathcal{A}_{\sigma_{T_2T_1}}(a_1)$, which means the change of the assets of user a_1 must be different given two transaction execution orders, i.e., T_1T_2 and T_2T_1. The assets of a_1 can only be changed in the profiting nodes of xSDG in the execution of T_1, and the difference of assets change is caused by the fact that the other transaction T_2 may write to the contract storage. Therefore, the necessary condition is that there must be an overlap between the influenced profiting nodes of f_2 and the reachable profiting nodes of f_1, i.e. $IP(f_2) \cap RP(f_1) \neq \emptyset$. Similarly, the necessary condition of victim loss, i.e., $\mathcal{A}_{\sigma_{T_1T_2}}(a_2) < \mathcal{A}_{\sigma_{T_2T_1}}(a_2)$, is $IP(f_1) \cap RP(f_2) \neq \emptyset$. □

Example. Consider the function pair $\langle \texttt{solve}, \texttt{solve}\rangle$, where $f_1 = f_2 = \texttt{solve}$ in Fig. 3. As illustrated previously, $IP(\texttt{solve}) = RP(\texttt{solve}) = \{$ⓓ$\}$. The necessary condition of the function pair being vulnerable is satisfied. Therefore, we will further analyze it using symbolic execution to validate it is vulnerable. In contrast, consider the function pair $\langle \texttt{solve}, \texttt{addFund}\rangle$ in Fig. 3. $IP(\texttt{addFund}) = \{$ⓓ$\}$, $RP(\texttt{addFund}) = \{$ⓘ$\}$, and $IP(\texttt{solve}) = RP(\texttt{solve}) = \{$ⓓ$\}$. The necessary condition $IP(\texttt{solve}) \cap RP(\texttt{addFund}) \neq \emptyset$ does not hold. If the transaction of `solve` is executed before `addFund`, the user calling `addFund` will still execute as expected, paying the exact amount of ethers to the contract as the user wants. The user is not affected by the front-running so this function pair is pruned and we will not symbolically execute it. □

4.1.3. Fast Pruning via Datalog. The key to checking the necessary condition (Definition 1) for a pair of functions $\langle f_1, f_2\rangle$ is to interpret the *reachable profiting nodes* and *influenced profiting nodes* for each function. We use Datalog [15] to define rules and inductively interpret the $RP(f)$ and $IP(f)$ for each function f in xSDG. Datalog rules consist of two parts: *facts* and *induction rules*, with the former describing basic relations that are explicit before inference and the latter defining how new relations can be inferred from facts and other relations by the Datalog engine.

Relation	Description
$edge(n, u, l)$	edge $\langle n, u\rangle \in E$ and $\lambda(\langle n, u\rangle) = l$.
$profit(n)$	node n is a profiting node
$rp(n, u)$	node $u \in RP(n)$.
$ip(n, u)$	node $u \in IP(n)$.
$dep(n, u)$	node u is a profiting node and depends on node n.

Figure 4: The definitions of relations in Datalog, given an xSDG $G = \langle N, E, \lambda\rangle$.

$dep(n,u)$	:-	$dep(w,u), edge(n,w,l), l \in \{\texttt{c}, \texttt{d}\}$	①
$ip(n,u)$	:-	$dep(w,u), edge(n,w,l), l = \texttt{D}$	②
$ip(n,u)$	:-	$ip(w,u), edge(n,w,l), l \in \{\texttt{c}, \texttt{C}, \texttt{d}\}$	③
$rp(n,u)$	:-	$rp(w,u), edge(n,w,l), l \in \{\texttt{c}, \texttt{C}, \texttt{d}\}$	④
$dep(u,u)$	:-	$profit(u)$	⑤
$rp(u,u)$	:-	$profit(u)$	⑥

Figure 5: Inference rules of relations in Fig. 4

First, we extend our definition of *reachable profiting nodes* and *influenced profiting nodes* to each node in the xSDG. The *reachable profiting nodes* of the node n, denoted by $RP(n)$, is the set of profiting nodes reachable on G^* from node n. Let D denote the set of storage variable nodes that are reachable on G^* from node n, the *influenced profiting nodes* of the node n, denoted by $IP(n)$, is the set of profiting nodes reachable on G' from any $u \in D$. The definitions of G^* and G' are the same as those in Definition 3 and 4. Clearly, the *reachable profiting nodes* and *influenced profiting nodes* of a function f is equavilent to those of the entry node of f, e.g., in Fig. 3, $RP(\texttt{solve}) = RP($ⓐ$)$ and $IP(\texttt{solve}) = IP($ⓐ$)$.

Second, for each node n, we design rules to infer $RP(n)$ and $IP(n)$ inductively. Fig. 4 shows the Datalog relations we define on the xSDG. In addition to the relations corresponding to the edges in the xSDG, profiting nodes, reachable profiting nodes, and influenced profiting nodes, we also introduce a utility relation $dep(n, u)$. The $dep(n, u)$ relation represents that node u is a profiting node, and node u depends (either control dependency or data dependency) on node n. In Fig. 5, we present the rules to infer the relations in Fig. 4. The relations $edge(n, u, l)$ and $profit(n)$ are facts provided initially to start the inference. If a node u is a profiting node, then we infer the $dep(u, u)$ and $rp(u, u)$ relations for u itself (rule ⑤ ⑥). Then we inductively infer the relations for other nodes according to the edge labels. In rule ①, if there is an edge $\langle n, w\rangle$ with label `c` or `d`, which means node w depends on n, then all the profiting nodes that depend on w is inherited by node n. In rule ②, if there is an edge $\langle n, w\rangle$ with label `D`, then all the profiting nodes that depend on node w also depend on node n, which is a storage variable node. Thus, all these profiting nodes are part of the influenced profiting nodes of node n, i.e., $IP(n)$. Similarly, node n inherits the influenced profiting nodes of

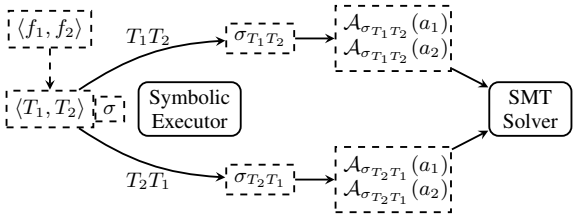


Figure 6: Symbolic validation of a function pair.

node w (rule ③) if the label of edge $\langle n, w\rangle$ is either `c`, `C`, or `d`. Rule ④ specifies that the reachable profiting nodes of w are also part of $RP(n)$ if node w control-depends on n or node n calls function at w.

Computation Complexity. Computing the *reachable profiting nodes* and *influenced profiting nodes* for each function is efficient. For a function f, the definitions of $RP(f)$ and $IP(f)$ (Definitions 3 and 4) describe the reachability relation between the entry node of f and profiting nodes on subgraphs of xSDG. Calculating the reachability relations from a single graph node is linear to the number of nodes. Thus, the complexity to calculate $RP(f)$ and $IP(f)$ for all $f \in \mathcal{F}$ in a contract group is $O(|N||\mathcal{F}|)$, where N is the set of all nodes in the xSDG and $\mathcal{F}$ is the set of all functions. Note that the number of functions $|\mathcal{F}|$ is significantly smaller than the number of nodes in xSDG, and the computation can be easily parallelized for each function. Thus, our static pruning is efficient.

4.2. Symbolic Validation

After the static pruning, we symbolically validate the remaining function pairs that satisfy the necessary condition to check whether they are truly vulnerable according to Definition 1. Fig. 6 shows the procedure of symbolic validation for a function pair $\langle f_1, f_2\rangle$. We first construct two independent symbolic transactions T_1 and T_2 for the two functions, respectively. The submitters of T_1 and T_2 are a_1 and a_2, respectively. Then we execute the two symbolic transactions sequentially in two different orders, i.e., T_1T_2 and T_2T_1, respectively, based on a symbolic blockchain state σ. During symbolic execution, we merge the symbolic values of the same variables in different paths to reduce the influence of path explosion. The rationale is that by merging symbolic variables in different paths, we do not solve constraints for each path. Instead, we collect the asset possessed by a transaction submitter as a single symbolic expression for each symbolic execution of the transaction. In the end, after symbolically executing two transaction orders, we derive the assets of the two submitters a_1 and a_2, i.e.,$\mathcal{A}_{\sigma_{T_1T_2}}(a_1)$, $\mathcal{A}_{\sigma_{T_1T_2}}(a_2)$, $\mathcal{A}_{\sigma_{T_2T_1}}(a_1)$, and $\mathcal{A}_{\sigma_{T_2T_1}}(a_2)$. Lastly, we use the SMT solver, Z3 [18], to check whether the predicates in Definition 1 are satisfiable. If true, we will report the function pair as vulnerable. How we compare users' assets in symbolic execution is explained in Appendix A.

Example. Suppose the contract in Fig. 3a has the following symbolic state, i.e., `hash`=h, `bounty`=b, `pubFund`=f. A symbolic blockchain state σ and two symbolic transactions $T_1 = \langle a_1, \texttt{solve}(s_1)\rangle$ and $T_2 = \langle a_2, \texttt{solve}(s_2)\rangle$ are created. We symbolically execute the two transaction orders separately, i.e., T_1T_2 and T_2T_1. Consider the scenario where T_1 is executed before T_2. In T_1, the assets of user a_1 (ethers balance) increase by $b + f$, since the reward is transferred to a_1 at Line 7, i.e., $\mathcal{A}_{\sigma_{T_1T_2}}(a_1) = \mathcal{A}_{\sigma}(a_1) + ite(s_1 = h, b+f, 0)$, where ite is the if-then-else symbolic expression which returns the second operand if the first operand is evaluated to be true or, otherwise returns the third operand. After the execution of T_1, storage variables `bounty` and `pubFund` are set to zero. Then when T_2 executes, the assets of a_2 remain the same, i.e., $\mathcal{A}_{\sigma_{T_1T_2}}(a_2) = \mathcal{A}_{\sigma}(a_2)$, since function `reward` returns zero. Similarly, when symbolically executing the transaction order T_2T_1, the assets of a_2 increase by $b + f$ while the assets of a_1 remain the same, i.e., $\mathcal{A}_{\sigma_{T_2T_1}}(a_2) = \mathcal{A}_{\sigma}(a_2) + ite(s_2 = h, b + f, 0)$, $\mathcal{A}_{\sigma_{T_2T_1}}(a_1) = \mathcal{A}_{\sigma}(a_1)$. In the end, we use an SMT solver to conclude that the condition in the vulnerability definition (Definition 1) is satisfiable when $s_1 = s_2 \wedge (b > 0 \vee f > 0)$. Therefore, the function pair $\langle \texttt{solve}, \texttt{solve}\rangle$ is vulnerable to front-running attacks. □

When developing our approach, we find that many contracts involve complex computations, which makes the SMT solver hard to solve the constraints within a reasonable time limit. Thus, we adopt the following heuristics to improve the performance of SMT solving. First, we use symbolic bit vectors with smaller bit-width to represent integers during symbolic execution. Many smart contracts involve computations on 256-bit integers, which are hard for SMT solvers to solve constraints. As such, we use symbolic 32-bit vectors to represent integers whose bit-width is larger than 32.

Second, we convert non-linear symbolic expressions to linear ones by assigning concrete values to symbolic values. For example, for a division operation whose both nominator and denominator are symbolic expressions, we will sample concrete values to assign to variables used in the denominator so that the division operation results in an output with linear expression. We only adopt this strategy when the SMT solving fails to finish within a time limit, i.e., when the SMT solving timeout, we will assign concrete values to non-linear symbolic expressions and try to use SMT solver to solve again.

Last but not least, many contracts use external math libraries (e.g., SafeMath provided by Openzeppelin [6]) to perform safe mathematical operations with safety checks (e.g., overflow/underflow check). These libraries include many branches that significantly increase the total number of paths that symbolic execution needs to explore. As such, we use operations provided by SMT solver to model the behaviors of these math libraries instead of symbolically executing them. Our heuristics may induce imprecision in the symbolic execution and thus may result in false positives and false negatives in vulnerability detection. However, our evaluation shows that our heuristics are effective in improving the performance of *Nyx* while preserving a high recall and precision.

4.3. Discussion

Nyx differs from previous front-running vulnerability detection techniques from two aspects. First, our approach

considers exploitability as the first-class citizen in the contract analysis. Previous works detect data races, neglecting whether the races are benign or exploitable by attackers. *Nyx*, however, uses a vulnerability detection oracle (Definition 1) defined from the aspect of attacker exploitation opportunities. Such oracle drastically increases the search space of analysis, so we further propose a static pruning technique to make the analysis scalable. Our static pruning is also based on a necessary condition of attack exploitability, instead of data races which still produce benign races and induce false alarms.

Second, *Nyx* is designed to support the analysis of cross-contract semantics. Previous empirical study [54] has pointed out that the program logic vulnerable to front-running attacks often lies across multiple contracts (the contract group analyzed by *Nyx*). Existing front-running vulnerability detectors cannot collectively analyze a contract group due to the enlarged search space and thus miss many vulnerabilities. Our static pruning is efficient to safely prune many search space and makes *Nyx* scalable to analyzing cross-contract semantics of DeFi applications.

Nyx requires the source code of the smart contracts under analysis to be available in that *Nyx* constructs a precise xSDG based on SlithIR [24] (Fig. 2), which is converted from contract source code by Slither [47]. This may limit the usage of *Nyx* only on contracts with source code available. However, we argue that focusing on source-code analysis is practically valuable. Our review of the top 50 DeFi projects on DefiLlama [5] indicates that all but only OpenSea [8] (in which only a small portion of code is closed-source) are open-source. Due to the trustless nature of blockchain, users are more willing to deposit their funds in open-source projects. In addition, the main application scenario of *Nyx* is for contract developers to analyze front-running risks in their contracts before deployment. Developers should have access to the contract source code. Therefore, it is practical for *Nyx* to require source code for analysis.

5. Evaluation

In this section, we evaluate our approach and answer the following research questions. **RQ1**: (Effectiveness) How effective is our approach compared to existing techniques? **RQ2**: (Performance) How efficient is our approach? **RQ3**: (Ablation Study) How effective are our static pruning and symbolic validation in terms of reducing search space and eliminating false positives, respectively? In addition, we also leverage *Nyx* to identify four developer-confirmed zero-day vulnerabilities, showing the real-world impacts of our work.

Benchmark. We adopt the front-running vulnerability benchmark collected by Zhang et al. [54] to evaluate our approach. This is the most recent benchmark specially designed for front-running vulnerabilities. It contains 513 real-world vulnerable contract groups. Each contract group has on average 144 functions (with a total of 74,096 functions and 1,453,578 LOC). Each contract group consists of one or more contracts closely coupled with each other to implement some functionalities (e.g., token swap). Each contract group is associated with a front-running vulnerability exploited in the Ethereum history. Each vulnerability is associated with a witness transaction pair. The witness transaction pair consists of the attack transaction that the attacker uses to launch a front-running attack on the victim transaction in the history. We can use the witness transaction pair as a ground truth to evaluate the recall of vulnerability detection. The detect recall and precision results are presented in Section 5.1.

It is important to note that several other front-running datasets have been collected by researchers, in addition to the aforementioned one. Torres et al. [46], for example, compiled a considerable dataset of attacks using pattern matching on execution traces. While Zhang et al. have demonstrated that the experimented benchmark [54] outperforms Torres et al.'s one in terms of attack diversity, an evaluation conducted on Torres et al.'s dataset is provided in Appendix E for interested readers. Furthermore, Perez and Livshits [36] collect an attack dataset of known vulnerable contracts, which are detected by our baseline techniques [1], [21]. However, we exclude this dataset from the evaluation, since its collection is restricted to those attacks that are detectable by the baseline techniques, which could bias the evaluation process.

Baselines. We evaluate *Nyx* against six existing techniques that support detecting front-running vulnerabilities, namely, Oyente [31], Mythril [34], Securify [32], Securify2 [44], Ethracer [30], TODler [35] and Sailfish [14]. Oyente and Mythril are symbolic execution-based tools. Securify is a static analysis tool using Datalog rules to infer potential vulnerabilities. Securify2 is a completely re-invented tool on top of Securify supported by the Ethereum official. Ethracer is a fuzzing tool, which tries to execute concrete transactions in different orders to find front-running issues. TODler is a static analyzer based on the data dependency extracted by the contract bytecode decompiler, Gigahorse [27]. Sailfish is a hybrid tool that combines rule-based static analysis with symbolic execution to find data races for front-running vulnerabilities. Another related work is NPChecker [51], leveraging static model checking to identify asset transfers with read-write hazards on contract global variables. However, NPChecker is not available to use (as also reported by Munir and Reichenbach [35]). We thereby do not include NPChecker as the baselines.

Experiment settings. We run *Nyx* as well as all baselines to detect each vulnerability in the benchmark. Many vulnerabilities in the benchmark are cross-contract, meaning that the labeled vulnerable location lies across multiple contracts. For every such vulnerability, we consider these multiple contracts as a contract group and use *Nyx* to analyze the contract group. However, none of the existing techniques support such cross-contract analysis. Therefore, we run all baselines on each contract of a contract group individually and collect their results for each contract. We set the timeout of each contract group analysis for each tool to three hours. The timeout of SMT solving in the symbolic validation of each function pair in *Nyx* is set to

15 minutes. We do not repeat the experiment since all the tools are deterministic. [2] The experiments are conducted on a machine with AMD Ryzen 3975x and 512GB RAM. Each analysis is executed in a single thread.

Metrics. We measure the recall of each tool using the ground truth provided by the benchmark. For *Nyx* and Ethracer, we consider a vulnerability in the benchmark to be successfully detected by the tool if it reports warnings on the function pair that the witness transaction pair invokes. For other baselines, they can only report a single function that may be affected by some other transactions. Therefore, we adopt an over-approximation strategy by considering a vulnerability is successfully detected by the tool if it reports warnings on any of the two functions invoked by the witness transaction pair.

In addition, we also evaluate the precision of each tool by manually inspecting a random sample of the detection results. For *Nyx* and Ethracer, which reports a pair of functions, $\langle f_1, f_2 \rangle$, as vulnerable, we check whether we can manually craft two concrete transactions t_1 and t_2 for f_1 and f_2, such that the submitter of t_1 can gain profits by front-running t_2, and t_2's submitter is harmed by the front-running attack. For other tools that only report a single function f as vulnerable, we try to manually identify another transaction f' such that an attacker can invoke f' to launch a front-running attack to a transaction of f. If such a f' exists, we consider the tool reports a true positive.

5.1. RQ1: Effectiveness

Recall. Table. 1 shows the detection recall of on the benchmark for each tool. The recall of each tool is calculated by the number of contract groups whose vulnerability is detected (column DE) divided by the total number of analyzed ones (column AN). *Nyx* achieves the highest recall, 90.19%, outperforming the baselines, among which the highest recall is only 57.55%. In the experiment result, we find that Securify2 and Sailfish fail in the analysis of many vulnerabilities (column ER in Table 1). We inspect the error in the analysis and find that the root cause is that newer versions of Solidity are not supported by these tools. We have reported the issue in the repositories of these two tools, but the issue is not fixed at present. Ethracer timeouts on a large number of vulnerabilities (column TO). The idea of Ethracer is similar to the symbolic validation of *Nyx*, i.e., enumerating all possible combinations of transactions calling different functions and checking if different execution orders result in different outcomes. The result of a large number of timeouts aligns with our discussion in Section 3.3, indicating that the search space for analyzing all possible pairs of functions is very large. In Section 5.3, we will show that our static pruning contributes a lot to reducing the search space. All baselines can barely detect vulnerabilities in the benchmark due to the limitations discussed in Section 3.2, i.e., lack of cross-contract analysis. In contrast, *Nyx* tackles the limitations and can detect front-running vulnerabilities with a high recall.

2. Ethracer, a fuzzing technique using SMT solvers to generate concrete transactions, is also deterministic across multiple runs.

TABLE 1: Detection Recall of Each Tool

Tool	Vulnerability Detection Results						
	TO	**ER**	**AN**	**DE**	**Recall**	**LOC**	**Time**
Oyente	0	0	513	0	0%	2,744.48	55.48s
Mythril	0	17	496	7	1.41%	2,750.18	98.28s
Securify	0	108	405	31	7.65%	2,286.73	701.04s
Securify2	0	503	10	0	0%	1,968.70	193.53s
Ethracer	178	1	334	13	3.89%	2,492.55	7,155.33s
TODler	0	162	351	202	57.55%	2555.56	27.26s
Sailfish	1	428	84	0	0%	1,260.87	32.83s
Nyx	6	79	428	386	**90.19%**	2,850.92	789.93s

TO: analysis timeout. **ER**: analysis error. **AN**: successfully analyzed. **DE**: vulnerability detected. **LOC**: average lines of code analyzed. **Time**: average analysis time.

False Negatives. We manually inspect the false negatives of *Nyx*. There are two reasons. One reason is that, the attack profits may not be transferred to the submitter of the attack transaction. The front-running attack profits may be transferred to an address variable different from the attack transaction submitter in the vulnerable contract. Although at runtime, the value of the transfer recipient variable is the same as the attack transaction submitter, it is hard for a static analysis tool like *Nyx* to know that the recipient of profits is the attacker. As a result, *Nyx* will falsely conclude that no profits are transferred to the attacker, and there is no exploitable front-running vulnerability. A case study of false negatives due to this reason is presented in Appendix C.

Another reason for false negatives is the imprecise modeling of contracts in our analysis. Some contracts use inline assembly or low-level contract calls to invoke a function in another contract. In the static analysis, it is hard to decide which contract and which function is invoked. As a result, these external function calls are missed by our analysis. If the external function call induces the vulnerability, *Nyx* will miss it.

Precision. Table. 2 shows the precision of each tool. The column Total shows the total numbers of reported warnings. For each tool, we randomly sample with 95% confidence level and 5% margin of error for manual inspection. If there are less than 50 warnings, we inspect all of them. The criteria of true positive (TP) has been described early in Section 5. The precision is calculated using the number of true positives divided by the total number of sampled warnings. Existing tools overall report 296 (51.12%) false positives out of the total 579 manually inspected warnings. In contrast, *Nyx* achieves high precision, 70.89%. Securify can also achieve a comparable precision, 68.00%, but the total number of warnings reported by Securify is much smaller than *Nyx*.

True Positives. To better understand the vulnerability uniquely detected by *Nyx* (not detected by other tools), we conduct a case study on the 224 TPs in the sampled warnings reported by *Nyx*. We summarize three categories of front-running vulnerabilities. We briefly introduce the three categories below and the details are presented in Appendix B with a few representative vulnerable contracts.

TABLE 2: Detection Precision of Each Tool

Tool	Manual Inspection Results			
	Total	Sampled	TP	Precision
Oyente	8	8	1	12.50%
Mythril	151	109	53	48.62%
Securify	50	50	34	68.00%
Securify2	0	0	0	-
Ethracer	120	92	31	33.70%
TODler	1743	315	151	47.94%
Sailfish	5	5	0	0%
Nyx	1767	316	224	**70.89%**

First, front-running vulnerabilities may induce price slippage in automated market maker (AMM) [33] contracts. AMM is a type of DeFi application that allows users to swap tokens and the token exchange rate is determined by calculating the ratio of reserves of two tokens being swapped. Attackers are able to swap the token in advance, change the token reserves of AMM contracts, and manipulate the exchange rate of victim transactions.

Second, front-running vulnerabilities may allow attackers to break the atomicity of multi-step on-chain actions. Many DeFi applications require users to perform a sequence of low-level actions (i.e., invocations to smart contracts) to complete a high-level functionality (e.g., remove liquidity from AMM [49]). Adversaries may insert transactions in the middle of ordinary users' action sequences, breaking the atomicity action sequences, and making profits.

Third, publicly obtainable profits in smart contracts can induce front-running attacks. Many contracts expose public profits (e.g., ERC20 tokens) or crypto-collectibles (e.g., NFTs) that can be obtained permissionlessly. The vulnerable contract lacks access control on the profit claim while the total amount of profits to claim is limited. A front-running attacker can execute transactions to obtain such profits before the targeted user, leading to losses for the victim since the expected profits are now unattainable.

Note that, in spite of a high precision and recall, *Nyx* reports a large number of warnings. This is because many smart contracts contain a lot of functions with similar logic but different function signatures. For example, in `UniswapV2Router02` contract of Uniswap V2 protocol [49], an implementation of AMM, there are nine different functions with the same functionality to swap one kind of token to another, e.g., ethers or ERC20 tokens. Different functions are provided to facilitate different use scenarios, e.g., to swap the exact amount of one token to another or to swap a certain amount of one token to the exact amount of another. *Nyx* finds that the token swap logic can be attacked and any pair of swap functions are vulnerable. All nine different functions have the same swap logic but with different names and parameter lists. Therefore, *Nyx* will report warnings for all of them. In this case, 45 warnings (pairs of swap functions) are reported for this same vulnerability in the swap logic. However, it is non-trivial for *Nyx* to automatically identify the same logic and remove such duplicates due to the lack of contract specifications. In our manual inspection of the sample, we find that 60.44% of the reported warnings are duplicate warnings in such contracts.

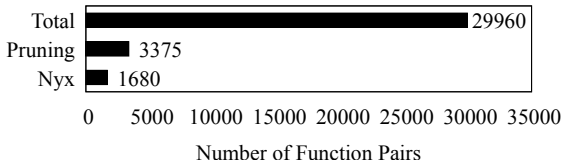


Figure 7: The total number of function pairs and the number of warnings reported by *Nyx*-P and *Nyx*.

TABLE 3: Detection Results of *Nyx*, *Nyx*-P, and *Nyx*-V.

Tool	Warnings	Recall	Precision	Avg. Time
Nyx	1767	90.19%	70.89%	789.93s
Nyx-P	3375	94.21%	35.07%	19.48s
Nyx-V	2207	92.67%	69.82%	2,319.37s

False Positives. We also investigate the false positives of *Nyx* and find three reasons. The first reason is that when performing external contract calls, the contract has implicit assumptions for the callee contract. For example, the caller assumes that a storage variable in the callee contract has a specific value at runtime. There are no assertions in the contract code encoding such assumptions. Thus, *Nyx* is unaware of the assumptions and raises false warnings for contract behaviors impossible at runtime. A case study of false positives due to this reason is presented in Appendix D.

The second reason for false positives is implicit constraints on storage variables. For example, the contract may be designed in a way that one storage variable always has the same value as the other storage variable, regardless of how the contract is invoked by transactions. However, there are no assertion statements checking this equivalence relationship. As a result, *Nyx* is not aware of such implicit constraints on storage variables and over-approximates the contract behaviors. A case study of false positives due to this reason is presented in Appendix D. It may be possible to infer such implicit constraints by analyzing all contract state-altering functions in advance. We leave such analysis as future work to further improve the precision of our approach.

Other false positives are caused by imprecise modeling of contract semantics during symbolic validation. There are some operations hard to model in symbolic execution, such as `abi.encode`/`abi.decode`, which encode/decode arbitrary structured data to/from a byte sequence, and cryptographic hash operations. When encountering such operations, *Nyx* treats the operation output as unconstrained symbolic values, which may induce false alarms.

Answer to RQ1: *Nyx* is effective to detect front-running vulnerabilities with high recall (90.19%) and precision (70.89%) on the benchmark of real-world smart contracts, which significantly outperforms existing tools.

5.2. RQ2: Efficiency

Table 1 also shows the analysis time of baselines and *Nyx* (column Time) averaged among contract groups that each tool successfully analyzed (column AN). Since the

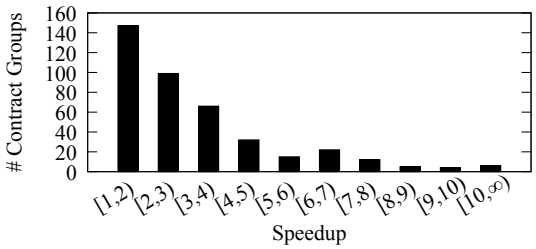


Figure 8: Speedup achieved with static pruning.

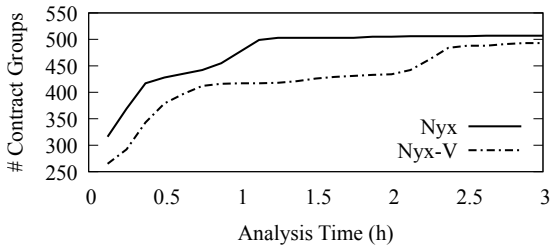


Figure 9: Number of contract groups that *Nyx* and *Nyx*-V finish analyze over time.

number of analyzed contract groups varies among tools, and the contracts may also vary in scale, we also show the number of lines of code (LOC) on average for each analyzed vulnerability (column LOC). Column Time shows the average analysis time for each contract group. On average, it takes 789.83s for *Nyx* to analyze each contract group, within which static pruning takes 9.30s and symbolic validation takes 754.93s, respectively. The results show that our static pruning is fast, while symbolic validation takes most of the analysis time. *Nyx* is slower than most of the baselines, but it is expected. As mentioned in Section 3.2, all baselines only analyze contracts individually without considering the cross-contract function calls. Thus, their search space is much smaller than *Nyx*, which supports cross-contract analysis. Analyzing contracts individually may make the analysis faster, but it causes a large number of vulnerabilities missed as shown in Section 5.1. In addition, *Nyx* takes one step further compared to baselines by ensuring the exploitability of the vulnerability, instead of only identifying data races. Checking whether a front-running is profitable for attackers and harmful for victims involves solving much more complex constraints. The slowdown in *Nyx* is a trade-off for higher precision. In fact, *Nyx*'s overall performance is comparable to the baseline with the best precision, Securify.

Answer to RQ2: The performance of *Nyx* is comparable to the baseline with the best vulnerability detection precision, i.e., Securify, and significantly faster than Ethracer. Not being faster than most of the baselines is expected since our approach analyzes a much larger search space to improve the detection recall and needs to solve more complex constraints to achieve high precision.

5.3. RQ3: Ablation Study

To get a better understanding of how the static pruning and symbolic validation contribute to the overall performance of *Nyx*, we perform an ablation study. We build two variants of *Nyx*: *Nyx*-P disables symbolic validation in *Nyx* and reports all function pairs after static pruning as warnings. *Nyx*-V disables static pruning in *Nyx* and directly applies symbolic execution on all possible function pairs. Table 3 shows the total number of warnings reported, the recall on the benchmark, the precision of reported warnings, and the average analysis time for each contract group. To obtain the precision of *Nyx*-P and *Nyx*-V, we sample 345 and 328 (95% confidence level and 5% margin of error) reported warnings for *Nyx*-P and *Nyx*-V, respectively, and manually check whether each warning is a false positive or not in the same way as for RQ1.

Both *Nyx* and the two variants achieve a high recall on the benchmark. However, *Nyx*-P has a low precision due to the over-approximation of front-running vulnerability in our static pruning. When applying symbolic validation on top of *Nyx*-P, *Nyx* is able to improve the precision to 70.89% (RQ1). *Nyx*-V achieves a similar precision as *Nyx*-V. However, *Nyx* is on average 2.93x faster than *Nyx*-V. Specifically, *Nyx*'s efficiency significantly surpasses that of *Nyx*-V when the complexity of contracts increases. *Nyx*-V cannot finish analyzing 22.43% contract groups within an hour, while *Nyx* struggles for only 2.34% of them (9.6-fold decrease). This enhancement is attributable to *Nyx*'s ability to drastically reduce the search space through static pruning. Fig. 9 shows the cumulative count of contract groups analyzed by *Nyx* and *Nyx*-V over time within the three-hour experiment time budget. *Nyx* finishes analysis for most contract groups within around one hour, while one-fifth of the contract groups cannot be analyzed by *Nyx*-V within the same duration.

To further understand the benefits of the static pruning in *Nyx*. We measure the number of function pairs that static pruning eliminates out of the entire search space. Fig. 7 shows the total number of function pairs that *Nyx*-V symbolically executes (i.e., the entire search space), the number of function pairs after static pruning (i.e., the function pairs *Nyx* symbolically executes), and the number of function pairs that *Nyx* reports in the end. There are, in total, 29,960 function pairs in contracts that contain vulnerabilities in the benchmark, i.e., the entire search space. After static pruning, there are only 3,375 function pairs left, reducing the search space by 88.73%. However, as aforementioned, there are a lot of false positives in the 3,375 function pairs. Symbolic validation further eliminates 1,695 unexploitable function pairs. We also measure the speedup that static pruning helps *Nyx* achieve. For a contract group in the benchmark, the speedup is calculated by the analysis time of *Nyx*-V divided by the analysis time of *Nyx*. Fig. 8 shows a histogram diagram of speedup for all contract groups in the benchmark.

Anaswer to RQ3: Static pruning and symbolic validation play important roles to reduce search space and improve the precision of detecting front-running vulnerabilities.

5.4. Real-World Impact

To show how our approach can help improve the security of smart contracts, we leverage *Nyx* to audit real-world contracts. By the time of writing, we have found four zero-day vulnerabilities in four DeFi projects, posing risks to over $320M on-chain funds. All the identified vulnerabilities are reported to contract developers, who confirm and fix them immediately. We are in total awarded a substantial bug bounty of $10268.30, underscoring the effectiveness and real-world significance of *Nyx*. One bug allows front-running attackers to progressively steal all deposited funds. Another bug allows denial of service attacks, where any valid user action is blocked for a substantial period of time. The third bug may induce malicious actors to claim ownership of the entire project, thereby placing all user assets at risk. The last finding allows malicious actors to take advantage of the reward system by front-running and unfairly claiming rewards. This discourages honest users from updating the bucket exchange rates and contributing to the system.

6. Related Work

Understanding front-running in smart contracts. Researchers have conducted extensive empirical studies to understand the front-running attacks. Sergey and Hobor [42] first consider smart contracts as shared objects in the concurrent invocation of transactions and point out the root cause of a variety of vulnerabilities in contracts from the concurrency perspective, including front-running. Daian et al. [17] propose Maximal Extractable Value (MEV) to measure the profits an attacker can make by manipulating transaction orders and reveal the fact that various front-running bots exist competing with each other for front-running profits. Being aware of the prevalent front-running attacks on the blockchain, Torres et al. [46], Perez and Livshits [36], Qin et al. [40], and Flashbots [25] conduct measurement studies and find that front-running attacks are pervasive and attackers have obtained a large amount of profits. To understand the vulnerable smart contract code that induces front-running attacks, Eskandari et al. [22], Durieux et al. [19], Ghaleb and Pattabiraman [26], Zhang et al. [56] and Zhang et al. [54] construct datasets of vulnerable smart contracts, which can also be used as benchmarks to evaluate vulnerability detection techniques. Front-running can also be leveraged as protection from other attacks. Zhang et al. [55] and Qin et al. [38] propose novel techniques to synthesize front-running attacks to attack malicious exploitation (e.g., reentrancy) on vulnerable contracts to recuse funds at risk. However, such techniques mimic existing attacks and require an already-exploited vulnerability. They have different application scenarios from *Nyx*, which detect previously unknown vulnerabilities in smart contracts.

Detecting front-running vulnerability. Researchers have made efforts to pinpoint the smart contract code patterns of front-running vulnerabilities, including transaction order dependency [31], event ordering bugs [30], and state inconsistency bugs [14]. Various techniques have been proposed to detect such code patterns in smart contracts off-chain (i.e., before contracts are deployed). Oyente [31] uses symbolic execution to detect front-running vulnerabilities by finding two execution paths that result in different ether transfer flows. Securify [48] adopts a datalog-based static analysis to identify ether transfers using data flowed from contract storage variables as vulnerable. NPChecker [51] uses taint analysis to find read-write hazards in smart contracts with model checking technique. Sailfish [14] also identify data races that affect ether transfers as vulnerabilities. Ethracer [30] generates concrete transactions using SMT solvers and executes them in different orders to check if there are races between transactions. All these techniques do not consider the exploitability of front-running vulnerabilities and cannot perform cross-contract analysis (discussed in Section 3.2). They have shown to have high false negative and false positive rates in our experiments (Section 5.1), and *Nyx* outperforms these tools by mitigating their limitations. In addition, there are on-chain contract analyzers, which rely on the historical transactions of the target contract to do vulnerability detection. IcyChecker [53] replays historical transactions in different execution orders and checks if the resulting blockchain state is different or not. Similar to Ethracer, IcyChecker also does not consider the exploitability of races between transactions and may report many false positives. On-chain contract analyzers have different application scenarios than *Nyx* since they can only analyze contracts already deployed on the blockchain and heavily rely on a fruitful transaction history of the contracts. Therefore, We do not consider on-chain contract analyzers as appropriate baselines to our approach.

7. Conclusion

We propose *Nyx* to detect exploitable front-running vulnerabilities. To mitigate the challenge of enlarged search space of checking exploitability, we design an innovative static pruning approach to prune the search space effectively. We adopt SMT solver to identify exploitable vulnerabilities in symbolic execution using our proposed oracle. The evaluation shows that *Nyx* outperforms existing tools and has also identified four developer-confirmed zero-day vulnerabilities.

Acknowledgment

This work was supported by National Natural Science Foundation of China (Grant No. 61932021), Hong Kong Research Grant Council/General Research Fund (Grant No. 16205821), Hong Kong Research Grant Council/Research Impact Fund (Grant No. R5034-18), Natural Sciences and Engineering Research Council of Canada (Grant No. RGPIN-2022-03744), Natural Sciences and Engineering Research Council of Canada (Grant No. DGECR-2022-00378), and Science and Technology Innovation Committee Foundation of Shenzhen (Grant no. ZDSYS20210623092007023).

References

[1] Securify v2.0. SRI Lab, ETH Zurich, July 2022.

[2] Block timestamp manipulation attack. https://cryptomarketpool.com/block-timestamp-manipulation-attack/, 2023.

[3] Code4rena | keeping high severity bugs out of production. https://code4rena.com/, 2023.

[4] Cryptokitties | collect and breed digital cats! https://www.cryptokitties.co/, 2023.

[5] Defillama. https://defillama.com/, 2023.

[6] Math - openzeppelin docs. https://docs.openzeppelin.com/contracts/2.x/api/math, 2023.

[7] The merge | ethereum.org. https://ethereum.org/en/roadmap/merge/, 2023.

[8] Opensea, the largest nft marketplace. https://opensea.io/, 2023.

[9] Openswap: Spot price queue. https://docs.openswap.xyz/#/technologies/liquidity-queue, 2023.

[10] Stableswap - efficient mechanism for stablecoin liquidity. https://classic.curve.fi/files/stableswap-paper.pdf, 2023.

[11] Tellor oracle protocol - transparent & permissionless. https://tellor.io/, 2023.

[12] Uniwhale: Unique oracle design. https://docs.uniwhale.co/unique-oracle-design, 2023.

[13] Weak block-based prng in solidity. https://blog.solidityscan.com/weak-block-based-prng-in-solidity-f29e089de594, 2023.

[14] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Cristopher Kruegel, and Giovanni Vigna. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. In *2022 IEEE Symposium on Security and Privacy (SP '22)*, pages 1235–1252. IEEE Computer Society, January 2022.

[15] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, March 1989.

[16] Jacques Dafflon, Jordi Baylina, and Thomas Shababi. EIP-777: Token Standard. https://eips.ethereum.org/EIPS/eip-777, April 2023.

[17] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges. *arXiv:1904.05234 [cs]*, 2019.

[18] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[19] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*, pages 530–541, New York, NY, USA, June 2020. Association for Computing Machinery.

[20] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. EIP-721: Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721, April 2023.

[21] Enzyme Finance. Enzymefinance/oyente. Enzyme Finance, June 2021.

[22] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. SoK: Transparent dishonesty: Front-running attacks on blockchain. In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security (FC '19)*, Lecture Notes in Computer Science, pages 170–189, Cham, February 2019. Springer International Publishing.

[23] Etherscan. Deplay. https://etherscan.io/address/0xe34443095f78099675b165f07559e9b48450c77e, May 2023.

[24] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB '19)*, pages 8–15, New York, NY, USA, May 2019. Association for Computing Machinery.

[25] Flashbots. MEV Explore. https://explore.flashbots.net/, April 2023.

[26] Asem Ghaleb and Karthik Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*, pages 415–427, New York, NY, USA, July 2020. Association for Computing Machinery.

[27] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, ICSE 2019, pages 1176–1186, Montreal, QC, Canada, May 2019. IEEE.

[28] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, January 1990.

[29] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[30] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*, pages 363–373, New York, NY, USA, July 2019. Association for Computing Machinery.

[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, pages 254–269, New York, NY, USA, October 2016. Association for Computing Machinery.

[32] Alexander Mense and Markus Flatscher. Security Vulnerabilities in Ethereum Smart Contracts. In *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*, iiWAS2018, pages 375–380, New York, NY, USA, November 2018. Association for Computing Machinery.

[33] Vijay Mohan. Automated market makers and decentralized exchanges: A DeFi primer. *Financial Innovation*, 8(1):20, February 2022.

[34] Bernhard Mueller. Smashing Ethereum Smart Contracts for Fun and ACTUAL Profit. In *The 9th Annual HITB Security Conference in the Netherlands (HITBSecConf '18)*, Hack In The Box Security Conference, pages 1–54, Amsterdam, Netherlands, April 2018.

[35] Sundas Munir and Christoph Reichenbach. TODLER: A Transaction Ordering Dependency anaLyzER - for Ethereum Smart Contracts. In *2023 IEEE/ACM 6th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16, May 2023.

[36] Daniel Perez and Ben Livshits. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *30th USENIX Security Symposium (USENIX Security '21)*, pages 1325–1341. USENIX Association, August 2021.

[37] Lambda philogy, berndartmueller. Token change can be frontrun, blocking token. https://code4rena.com/reports/2022-07-juicebox#h-02-token-change-can-be-frontrun-blocking-token, May 2023.

[38] Kaihua Qin, Benjamin Livshits, Stefanos Chaliasos, Liyi Zhou, Dawn Song, and Arthur Gervais. The Blockchain Imitation Game.

[39] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying Blockchain Extractable Value: How dark is the forest?, December 2021.

```
1 contract AMM {
2  ERC20 tokenIn, tokenOut;
3  function swap(uint in) returns (uint out) {
4    // calculate output token amount
5    uint reserve0 = tokenIn.balanceOf(this);
6    uint reserve1 = tokenOut.balanceOf(this);
7    uint k = reserve0 * reserve1;
8    out = reserve1 - k / (reserve0 + in);
9    // swap
10   tokenIn.transferFrom(msg.sender, this, in);
11   tokenOut.transferFrom(this, msg.sender, out);
12  }}
```

Figure 10: An Automated Market Maker (AMM) contract that contains front-running vulnerability.

[40] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying Blockchain Extractable Value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP '22)*, pages 198–214. IEEE Computer Society, May 2022.

[41] Witek Radomski, Andrew Cooke, Philippe Castonguay, James Therien, Eric Binet, and Ronan Sandford. EIP-1155: Multi Token Standard. https://eips.ethereum.org/EIPS/eip-1155, April 2023.

[42] Ilya Sergey and Aquinas Hobor. A Concurrent Perspective on Smart Contracts. *arXiv:1702.05511 [cs]*, February 2017.

[43] Shipyard Software Inc. Clipper DEX. https://clipper.exchange/, May 2023.

[44] SRI Lab. Securify2/securify at master · eth-sri/securify2. https://github.com/eth-sri/securify2, April 2023.

[45] Sushi. SushiSwap. https://www.sushi.com, May 2023.

[46] Christof Ferreira Torres, Ramiro Camino, and Radu State. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium (USENIX Security '21)*, pages 1343–1359. USENIX Association, August 2021.

[47] Trail of Bits. Slither/slither at master · crytic/slither. https://github.com/crytic/slither, June 2023.

[48] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, CCS 2018, pages 67–82, New York, NY, USA, October 2018. Association for Computing Machinery.

[49] Uniswap. Decentralized trading protocol. https://uniswap.org/, December 2021.

[50] Fabian Vogelsteller and Vitalik Buterin. EIP-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20, April 2023.

[51] Shuai Wang, Chengyu Zhang, and Zhendong Su. Detecting nondeterministic payment bugs in Ethereum smart contracts. *Proc. ACM Program. Lang.*, 3(OOPSLA):1–29, October 2019.

[52] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf, 2020.

[53] Mingxi Ye, Yuhong Nan, Zibin Zheng, Dongpeng Wu, and Huizhong Li. Detecting State Inconsistency Bugs in DApps via On-Chain Transaction Replay and Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pages 298–309, New York, NY, USA, July 2023. Association for Computing Machinery.

[54] Wuqi Zhang, Lili Wei, Shing-Chi Cheung, Yepang Liu, Shuqing Li, Lu Liu, and Michael R. Lyu. Combatting Front-Running in Smart Contracts: Attack Mining, Benchmark Construction and Vulnerability Detector Evaluation. *IEEE Transactions on Software Engineering*, pages 1–17, 2023.

[55] Zhuo Zhang, Zhiqiang Lin, Marcelo Morales, Xiangyu Zhang, and Kaiyuan Zhang. Your exploit is mine: instantly synthesizing counterattack smart contract. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1757–1774, 2023.

[56] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying Exploitable Bugs in Smart Contracts. In *Proceedings of the ACM/IEEE 45nd International Conference on Software Engineering (ICSE '23)*, New York, NY, USA, May 2023. Association for Computing Machinery.

[57] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V. Le, and Arthur Gervais. High-Frequency Trading on Decentralized On-Chain Exchanges. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 455–472. IEEE Computer Society, Invalid Date.

Appendix A. Comparing Assets in Vulnerability Detection

When comparing the assets of user a on two different blockchain states σ_1 and σ_2, the user may possess different kinds of assets and $\mathcal{A}_{\sigma_1}(a)$ and $\mathcal{A}_{\sigma_2}(a)$ may not be directly comparable. Let k denote a specific kind of asset (e.g., ether or USDC token), and $\mathcal{A}^k_\sigma(a)$ denote the balance of asset k of user a. We use the following sufficient condition to approximate the comparison between $\mathcal{A}_{\sigma_1}(a)$ and $\mathcal{A}_{\sigma_2}(a)$:

$$\forall k \in \mathcal{K}, \mathcal{A}^k_{\sigma_1}(a) \geq \mathcal{A}^k_{\sigma_2}(a) \wedge \exists k \in \mathcal{K}, \mathcal{A}^k_{\sigma_1}(a) > \mathcal{A}^k_{\sigma_2}(a) \\ \implies \mathcal{A}_{\sigma_1}(a) > \mathcal{A}_{\sigma_2}(a)$$

where $\mathcal{K}$ is all the kinds of assets that users can possess.

Appendix B. Case Study for True Positives

In Section 5.1, we summarize three categories of front-running vulnerabilities uniquely detected by *Nyx* (not detected by other tools). For each category, we present a representative vulnerable contract simplified from real-world contracts in the benchmark [54].

Price slippage in Automated Market Maker (AMM) contracts. AMM is a kind of smart contract that supports automated swaps between two types of fungible tokens without the need to match buyers and sellers [33]. Fig. 10 shows one typical implementation of AMM adopted by many popular DeFi projects (e.g., Uniswap [49], SushiSwap [45], etc.). Function `swap` allows users to swap `in` amount of `tokenIn` to `tokenTo`. The token exchange rate is determined by invariant k, which is the multiplication of balances of the two tokens (i.e., token reserves) held by the AMM contract (lines 5-7). The contract determines the amount of swap output token by the principle that the invariant k should remain the same before and after the swap action. Qualitatively, the exchange rate decreases after each swap action from `tokenIn` to `tokenOut`. When an ordinary user submits a transaction (victim), an attacker can invoke function `swap` in advance (front-running attack), decreasing the exchange rate in the victim transaction (price slippage). As a result, the ordinary user swaps tokens at a lower exchange rate, suffering from loss. Note that the ordinary user's transaction further decreases the exchange rate from `tokenIn` to `tokenOut` (i.e., the reverse exchange rate increases). Therefore, the attacker can later swap back to `tokenIn` to make profits since the exchange rate of reverse

```
1 contract KittyCore {
2   function giveBirth(uint matronId) returns (uint
        kittenId) {
3     // Grab a reference to the matron in storage.
4     Kitty storage matron = kitties[matronId];
5     // Check that the matron is pregnant, and that
          its time has come!
6     require(_isReadyToGiveBirth(matron));
7
8     // Make the new kitten!
9     address owner = kittyIndexToOwner[matronId];
10    kittenId = _createKitty(matronId, matron.
          siringWithId, owner);
11    msg.sender.send(autoBirthFee);
12  }}
```

Figure 12: A vulnerable contract simplified from CryptoKitty [4].

```
1 contract Tellor {
2   function submitMiningSolution(string memory nonce,
        uint[5] requestIds, uint[5] values) external
        {
3     require(correctMiningSolution(requestIds, values)
        );
4     _newBlock(nonce, requestIds);
5   address[5] memory miners = requestDetails[
        _requestIds[0]].minersByValue[
        _timeOfLastNewValueVar];
6     //pay Miners Rewards
7     _payReward(miners, _previousTime);
8     _adjustMiningDifficulty(nonce);
9   }}
```

Figure 13: A false negative example. The example is simplified from Tellor Oracle protocol contract [11].

```
1 contract LPToken is ERC20 {
2   ERC20 token0, token1;
3   function burn(address to) returns (uint amount0,
        uint amount1) {
4     // The LPToken to burn should have been
          transferred to this contract.
5     uint burn_amount = self.balance(this);
6     // Calculate the amount of token0 and token1 to
          redeem.
7     uint balance0 = token0.balanceOf(this);
8     uint balance1 = token1.balanceOf(this);
9     amount0 = burn_amount * balance0 / totalSupply;
10    amount1 = burn_amount * balance1 / totalSupply;
11    // Burn LPToken and transfer token0 and token1.
12    _burn(burn_amount);
13    token0.transferFrom(this, to, amount0);
14    token1.transferFrom(this, to, amount1);
15  }}
```

Figure 11: A vulnerable contract simplified from the LP token of UniswapV2 [49] and SushiSwap [45].

swap increases. Such front-running vulnerability is due to the flaw in the design of the AMM contract. Researchers and DeFi community are exploring better designs of AMM contracts (e.g., StableSwap in Curve [10], spot price queue in OpenSwap [9], oracle-based exchange rate in Uniwhale [12], etc.) that have little to zero token exchange price slippage (i.e., attackers have no way to manipulate the exchange rate of a victim transaction) to mitigate such front-running vulnerability.

Atomicity breaking in multi-step on-chain actions. Fig. 11 shows a vulnerable contract implementing the ERC20 token: LP. The function `burn` allows users to burn LP tokens they hold and redeem for another two types of tokens (`token0` and `token1`) proportionally (lines 8-11). To burn LP tokens, users have to perform two consecutive actions: 1) transfer a certain amount of LP tokens to the contract, and 2) call `burn` function specifying the address to which the redeemed `token0` and `token1` should be transferred (lines 15-16). The vulnerability exists in that an attacker may invoke the `burn` function after an ordinary user has finished the first action and before the user performs the second one. In the attack transaction, the contract mistakenly treats all its LP token balance (line 5) as the amount to burn by the attacker, even though the LP token originally belongs to the user instead of the attacker. Then, the redeemed tokens (lines 15-16) are transferred to an address specified by the attacker.

Preemption of publicly obtainable profits. Fig. 12 shows a snippet of a contract simplified from CryptoKitty [4]. CryptoKitty implements cartoon kitties as Non-Fungible Tokens (NFT) [20], where each kitty is unique with distinct features on the blockchain. Each kitty is created through a "pregnancy" process and after a period of time, the kitty is ready to give birth (line 6). The caller to the `giveBirth` function receives a reward for the birth of the kitty (line 12). Note that rth to a kitty as long as the kitty's birth is ready. However, only the first caller gets the reward since the kitty can only be born once. Attackers can front-run other users' invocations to `giveBirth` function to obtain the reward as profits since the contract enforces no access control, causing loss of rewards for the users. Daian et al. [17] also point out that such permissionlessly obtainable profits may also pose a threat to blockchain system (fee-based blockchain forking attack) in that malicious block miners may have the incentive to maliciously fork a blockchain if the profits made in the front-running attacks exceed the profits they can earn from honest block mining.

Appendix C. Case Study for False Negatives

We present a case where *Nyx* fails to detect the vulnerability because the attack profit is transferred to other accounts different from the attack transaction submitter. Fig. 13 shows a simplified version of a vulnerable contract that *Nyx* fails to detect. The contract `Tellor` implements a blockchain-like logic. The function `submitMiningSolution` can be called by a block miner who calculates the correct solution (`requestIds` and `values`) to get the reward for block mining at Line 7. The contract is vulnerable to front-running in a way similar to the vulnerability in Fig. 3 that only the first miner submitting the correct solution can get the block reward. Attackers can front-run victims' transactions to obtain the current block reward as profits while the victims lose the reward. Note that in this example, the recipient of the block reward (the attack profit) is obtained from the contract storage, instead of the transaction submitter. At runtime, the attacker's account is one of the miners at Line 5 to whom the reward is paid. However, during static analysis, it is hard

```
1 contract ClipperExchange {
2   ERC20 theExchange;
3   function withdrawAll() public {
4     require(theExchange.balanceOf(msg.sender) ==
          theExchange.totalSupply());
5     theExchange.burn(msg.sender, amount);
6     uint amount = address(this).balance;
7     msg.sender.transfer(amount);
8   }}
```

Figure 15: A false positive example due to the implicit constraints in contract state. The example is adapted from Clipper Exchange protocol [43].

```
1 contract UniswapV2Router {
2   function addLiquidity(ERC20 tokenA, uint amountA,
        address to) external {
3     UniswapV2Pair pair = UniswapV2Library.pairFor(
          factory, tokenA);
4     tokenA.transferFrom(msg.sender, to, amountA);
5     pair.mint(to);
6   }}
7 contract UniswapV2Pair {
8   uint reserve0; ERC20 token0;
9   function mint(address to) external lock returns (
        uint liquidity) {
10    uint balance0 = token0.balanceOf(this);
11    uint amount0 = balance0 - reserve0;
12    uint liquidity = computation(amount0);
13    _mint(to, liquidity);
14    reserve0 = balance0;
15  }}
```

Figure 14: A false positive example due to the implicit contract assumption. The example is adapted from Uniswap V2 protocol contracts [49].

for *Nyx* to know that the miners obtained at Line 5 contain the attack transaction submitter. As a result, *Nyx* falsely concludes that no profits are transferred to the attacker, and the vulnerability is missed.

Appendix D. Case Study for False Positives

We present two cases that *Nyx* report false positives due to *implicit assumptions for external contracts* and *implicit constraints on storage variables*, respectively.

Implicit assumptions for external contracts. Fig. 14 shows an example, where *Nyx* falsely reports function pair ⟨`addLiquidity`, `addLiquidity`⟩ as vulnerable. In the `addLiquidity` function, the transaction submitter transfers a certain amount of `tokenA` to the `UniswapV2Pair` contract at Line 4, and `UniswapV2Pair` mints liquidity tokens in return to the submitter at Line 13. The amount of liquidity tokens to be minted is calculated based on the amount of `tokenA` transferred. The `addLiquidity` function is not vulnerable since the number of liquidity tokens the user receives remains the same regardless of whether the transaction is front-run or not. However, during symbolic validation, *Nyx* has no way to know that `amount0` at Line 11 is always equal to `amountA` at Line 4. There is an implicit assumption that `token0` at Line 10 is the same as `tokenA` at Line 4. Thus, `amount0`, the balance difference calculated at Line 11, is guaranteed to be the amount of `tokenA` transferred at Line 4. However, during symbolic validation, *Nyx* has no way to know that the `token0` and `tokanA` are the same. This is because the value of `token0` is dynamically set in the contract deployment transaction, while the `pairFor` function at Line 3 only obtains the address of the already-deployed `UniswapV2Pair` contract. As a result, the variable `amount0` is treated as an unconstrained symbolic value, and *Nyx* falsely concludes that the liquidity token the transaction submitter receives can be manipulated by another transaction via a front-running attack. For a static analyzer like *Nyx*, such implicit assumptions are hard to infer during analysis.

Implicit constraints on storage variables. Fig. 15 shows an example in the Clipper Exchange protocol [43], where *Nyx* raises false alarms. The `withdrawAll` function is used to burn all the ERC20 tokens `theExchange` ever supplies (Line 5) and transfer all the balance of the current contract to whoever calls the function (Line 7). *Nyx* falsely reports the function pair ⟨`withdrawAll`, `withdrawAll`⟩ as vulnerable since whoever calls the function first will get all the balance of the contract as profit at Line 7, and other later callers will get zero. However, the function has a precondition that the caller must process the entire `totalSupply` of `theExchange` (Line 4). There is an implicit constraint that if any user processes the total supply of `theExchange` tokens, other users must all have zero balance. Thus, at runtime, only one user can call this function and no other attackers can pass the precondition at Line 4. However, *Nyx* has no idea about this implicit constraint. During symbolic validation, *Nyx* considers that multiple users may process the amount of `theExchange` tokens equal to `totalSupply`, and it is possible that a malicious user can front-run to call this function and take the contract balance as profits, while other callers receive nothing at Line 7.

Appendix E. Evaluation of *Nyx* on other dataset

In addition to the benchmark collected by Zhang et al. [54] that we used in the Section 5, Torres et al. [46] also collect a large dataset consisting of 199,724 attacks using pattern matching on historical transaction execution traces. However, their dataset is not ideal for our experiment since they collect attacks instead of vulnerable contracts. The authors do not provide the ground truth on which function is vulnerable to front-running and the dataset may contain multiple attacks due to the same vulnerability in the same DeFi application. Nevertheless, to mitigate the potential threat to validity that evaluation *Nyx* only on one benchmark may draw biased conclusions, we construct another benchmark by randomly sampling 513 attacks (the same size as the benchmark used in Section 5) from this dataset. To conduct experiments, we consider all contracts executed in the victim transaction of each attack as a contract group to be analyzed by *Nyx*. We consider *Nyx* successfully detects the underlying vulnerability of an attack if a warning is generated for the pair of functions called by the attack and victim transaction of the attack, respectively. As a result, *Nyx* is able to report vulnerabilities for 492 out of 513 attacks, giving a recall of 95.9%. This indicates that *Nyx* is also effective in Torres et al.'s dataset.

Appendix F.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

F.1. Summary

This paper introduces a novel technique and tool, Nyx, to detect front-running attacks in smart contracts with static analysis and symbolic execution. Nyx examines all possible contract function pairs and proposes a pruning technique to handle the huge search space.

F.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science

F.3. Reasons for Acceptance

1) The paper proposes a tool, called Nyx. Nyx outperforms the SOTA in terms of accuracy.
2) Nyx finds four zero-days in real-world DeFi applications.