HOUSTON: Real-Time Anomaly Detection of Attacks against Ethereum DeFi Protocols

Dongyu Meng[1*], Fabio Gritti[1*], Robert McLaughlin[1], Nicola Ruaro[1],
Ilya Grishchenko[2], Christopher Kruegel[1], and Giovanni Vigna[1]
[1]University of California, Santa Barbara
{dmeng, degrigis, robert349, ruaronicola, chris, vigna}@ucsb.edu
[2]University of Toronto
ilya.grishchenko@utoronto.ca

***Abstract*—As decentralized finance (DeFi) continues to innovate the financial system, the security of its building blocks remains a critical concern to its large-scale adoption. In DeFi, the stakes are exceptionally high, marked by recurring instances of financial losses totaling millions of dollars every week. All major blockchain-based financial applications (i.e., DeFi protocols) are built from – and interact with – programs known as *smart contracts*. While many security tools have been developed to identify specific classes of vulnerabilities (e.g., reentrancy) in individual smart contracts, considerably less effort has been invested in automatically identifying – in real time – attacks against DeFi protocols.**

In this paper, we propose a novel approach for real-time, generic, explainable identification of attacks against DeFi protocols. Specifically, we identify potentially risky transactions *without* relying on any known vulnerability patterns. Our approach, implemented in HOUSTON, first automatically identifies the set of smart contracts that together implement a DeFi application and then, while monitoring new relevant transactions, builds and updates custom anomaly-detection models. Our models include information about typical execution paths (control flows) as well as information about how the protocol processes data, captured as likely invariants between the contract functions' arguments and storage variables. HOUSTON offers explainable warnings that can be used for attack triaging.

We evaluated HOUSTON on a large corpus of over 22 million transactions, covering 115 DeFi incidents. In our experiments, HOUSTON achieved a detection true-positive rate of 94.8% while maintaining a low false-positive rate. When compared with state-of-the-art anomaly detection systems, HOUSTON achieves a higher number of true positives and lower false-positive rates. Finally, we deployed HOUSTON in a real-world setting, where it demonstrated real-time monitoring capabilities on commodity hardware while sustaining high accuracy.

I. INTRODUCTION

Decentralized Finance (DeFi) drives a revolutionary paradigm shift in the financial landscape. Unlike traditional financial systems that rely on centralized authorities (e.g., banks), DeFi leverages blockchain technology to establish a transparent and decentralized financial ecosystem [60]. Ethereum is one of the most widely used blockchains (with a market cap of 450 billion USD at the time of writing [90]) and arguably the most popular environment for deploying DeFi applications. Hundreds of DeFi applications have been developed on Ethereum to provide financial services with a Total Value Locked (TVL) of almost 82 billion USD [19]. At their core, DeFi applications (or *protocols*) are implemented through one or more *smart contracts*. These smart contracts are immutable programs stored on the blockchain as bytecode and executed by the Ethereum Virtual Machine (EVM) [87]. For instance, a user may request to borrow a certain amount of cryptocurrency from a DeFi lending protocol and provide collateral in a different cryptocurrency as a security deposit. To fulfill this request, the protocol might involve several contracts, such as the "risk-manager contract" (to check for the safety of the borrower's position), the "oracle contract" (to report the correct price for the exchanged digital goods), and the "vault contract" (to transfer the funds).

While DeFi holds substantial promise, its proponents and users have learned a harsh lesson: Like all software, smart contracts have bugs, and these bugs can be (and have been) exploited by attackers to launch attacks that directly steal money from DeFi protocols [67], [15], [16]. At the time of writing, DeFi hacks have resulted in financial losses of more than 5 billion USD [8], [79]. Some of these stolen funds are known to sustain the activities of criminal groups [27] and dangerous nuclear programs [42]. A smart contract can be affected by multiple types of vulnerabilities [95]. Some of these vulnerabilities are basic implementation flaws (such as integer overflows and reentrancy). These bugs are fairly well-understood by the security community, and in recent years, both academia and industry have proposed a plethora of security tools to identify them [44], [29], [4], [56], [5], [33], [17], [35], [64]. Unfortunately, despite the use of these tools to secure smart contracts, attacks continue to occur. One reason is that certain bugs are tightly coupled to the business logic of individual DeFi protocols. The identification of such logic bugs requires a deep understanding of the protocol's functionality, which is beyond the reach of most existing automated analysis systems [95].

An alternative approach to identifying (and fixing) bugs is to detect attacks against vulnerable contracts after deployment.

[*] Equal contribution.

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241534
www.ndss-symposium.org

To this end, researchers have proposed systems that monitor blockchain transactions. Some systems examine the behavior of transactions that have already been executed with the idea of identifying attacks *a posteriori* [88], [74]. Other systems aim to detect attack transactions before they are committed to the blockchain (while they are still waiting in the public queue) [30]. When such pending attacks are detected, one can attempt to preempt (front-run) the attack transaction with a “defensive” transaction that rescues the funds held by the victim [93], [58], [89].

There are two main approaches for detecting attack transactions: *pattern-based techniques* and *anomaly-based techniques*. Pattern-based techniques leverage rules or heuristics that are applied to the execution traces of transactions [59], [91]. For example, DeFiRanger [88] introduces money-flow patterns that capture symptoms of price manipulation attacks. These techniques are generally precise and tend to produce few false alerts, but they are limited to detecting attacks that match known patterns. A second set of detection systems relies on anomaly detection [30], [54]. Their goal is to identify transactions that result in execution traces, data flows, or other aspects that are sufficiently different from those of benign transactions. Anomaly-based techniques are vulnerability-agnostic and can detect attacks that produce previously unseen anomalous behaviors. However, existing systems often lack explainability and typically generate a substantial number of false positives.

In this paper, we present HOUSTON, a novel anomaly detection system that performs real-time identification of attack transactions that target DeFi applications on Ethereum. Our key insight and contribution is the automated extraction and modeling of *semantically meaningful signals* from protocol executions. That is, we analyze transaction execution traces and focus on behaviors that reflect the core functionalities of a DeFi application. By doing so, we establish a robust baseline for normal behavior, enhancing our ability to detect outliers that are more likely to be attacks.

We model the behavior of a DeFi protocol by analyzing how functions within the protocol are historically invoked and how the persistent storage of the protocol’s contracts is manipulated. More precisely, our system comprises two distinct models, which are built and continually refined by monitoring live transactions involving the protocol under analysis: The *Interaction Model* and the *Invariant Model*. Here, we use the term “model” not in the machine learning sense, but to describe two purpose-built components: The *Interaction Model* is a control-flow-centric model that captures patterns of function invocations that perform sensitive operations. The *Invariant Model* is a data-state-centric model that infers relationships, in the form of *likely invariants*, between the inputs of smart contract functions and the values of the permanent storage of the contract. During live monitoring, HOUSTON continually consults and updates each model to examine the incoming stream of transactions. If a transaction is flagged as anomalous by any model, it is then escalated for evaluation by human experts or passed to automated response systems to mitigate the attack (e.g., by front-running the transaction [93], [58], [89]). Transactions deemed non-anomalous are instead utilized to refine the models, enhancing their accuracy and reducing the likelihood of future false positives. Importantly, assessments made by the models are executed rapidly, concluding well within the block generation time, before the possible inclusion of the attack transaction in the blockchain.

In summary, this paper makes the following contributions:

- We introduce models to capture the normal behavior of DeFi applications and leverage them to quickly detect *when* and *how* a DeFi protocol is under attack.
- Based on our models, we develop HOUSTON, an anomaly-detection system that can be *automatically* tailored to protect *any* given DeFi protocol on Ethereum.
- We compile 115 DeFi protocol incidents into a comprehensive benchmark, drawing from public datasets and incident reports. For each attack, we provide the address of the contract being attacked, the block and date of the incident, and the hash of the attack transaction. To the best of our knowledge, this is the most precise publicly available dataset on DeFi attacks.
- We compare the results of HOUSTON (94.8% true-positive rate, 0.16% false-positive rate) against state-of-the-art systems and demonstrate its superior performance.
- We deploy HOUSTON as a real-time monitoring solution and evaluate its performance while monitoring 20 DeFi protocols, showing that real-time analysis is achievable with modest hardware requirements.

II. BACKGROUND

A. Smart Contracts & DeFi

Smart Contracts. The Ethereum blockchain supports executing programs known as *smart contracts* directly on-chain. Every smart contract is associated with a unique address, and its deployed code is immutable. Anyone can send a transaction to a smart contract by paying a small (gas) fee. When a transaction is sent, the contract’s bytecode is executed by the Ethereum Virtual Machine (EVM) [23]. Smart contracts are typically written in a high-level language, such as Solidity [78], and then compiled into EVM bytecode.

Storage Layout. During execution, contracts use two types of data storage: persistent storage and volatile memory. Volatile memory is a form of temporary data storage that exists only for the duration of a single transaction, enabling short-lived data manipulation without incurring the high costs of on-chain storage. On the other hand, persistent storage serves as the contract’s long-term memory, retaining the values of the state variables across transactions [71].

Function Calls. To provide any functionality, a smart contract must expose one or more functions that are publicly callable. These functions can be invoked by a transaction that is initiated by a blockchain user (i.e., an externally owned account) or by another contract. When invoking the function of a smart contract, the caller specifies an input (`CALLDATA`), which contains the function identifier and its corresponding arguments. This enables the compiled contract to direct the execution to the appropriate function.

DeFi Protocols. DeFi Protocols provide financial services, such as lending, borrowing, and trading of crypto assets. Examples of DeFi protocols on Ethereum include MakerDAO [49] (a credit platform) and Uniswap (a token exchange) [85]. DeFi protocols are implemented with one or more interconnected smart contracts, each designed to handle specific tasks. For example, at the time of writing, MakerDAO comprises ∼415 smart contracts.

B. Anomaly Detection

Anomaly detection is an analysis approach that focuses on identifying items (or patterns) in a dataset that deviate significantly from the norm [9]. Anomalies may indicate issues varying in severity, ranging from full system compromise to simple unexpected edge cases. For example, if a smart contract sells digital artwork, (benign) execution traces will show that the code *always* checks to ensure that payment has been received before it transfers away ownership of the artwork. In this example, a possible anomaly would be identifying, within a trace, an ownership transfer *without* a check for payment. Anomaly detection has been successfully used for numerous security applications in different fields [43], [77].

III. Motivation

HOUSTON helps administrators of DeFi protocols to protect their smart contracts against attacks. An attack consists of one or more malicious transactions that an adversary uses to trigger unintended functionality in a DeFi protocol, typically with the intent to steal assets or disrupt its operations. Frequently, an attack will exploit some smart contract vulnerability, either a basic implementation bug or a logic flaw, to perform actions that the protocol developers did not expect.

HOUSTON is an *off-chain* monitoring solution that provides *real-time* attack detection by continuously monitoring transactions and identifying anomalies. Our system can analyze both transactions that are pending (in the *mempool*) and confirmed (appear in a block). The mempool is a publicly observable queue of transactions that have been submitted to the blockchain but have not yet been included in a block. If an attack transaction is submitted to the mempool, HOUSTON can simulate the execution of the transaction before it is confirmed, and raise an alert if an anomaly is detected. This alert can be used to prevent the attack: For example, administrators could implement an automatic emergency pause mechanism triggered by the alert [96], or the alert could drive automated systems that front-run suspicious transactions [89], [58], [93].

Unfortunately, transactions can bypass the mempool by using a private transaction relay, which forwards transactions to block producers in secrecy [1]. These transactions are generally referred to as *private transactions*, as they are first observed by the public only when they are announced as confirmed in a block. While originally designed to improve the privacy of the blockchain, private transactions are commonly abused to launch stealth attacks [48]. In these cases, our system cannot detect the attack before it is included in a block without cooperation from the private relay. However, even in such cases, an anomaly detection system can provide value. For instance, the attack against the `Revest` protocol [3] was carried out with four different private transactions (sent in two different blocks) over ∼17 minutes. HOUSTON can detect the first attack transaction (included in a block) and alert the protocol administrators. Thus, if HOUSTON had been deployed, the administrator would have had time to react and (potentially) mitigate the subsequent attacks.

Threat Model. DeFi is sometimes referred to as the "Wild West of Crypto" [12]. This is because the ecosystem is plagued by many threats and risks, including hacks, scams, and crypto-wallet thefts. However, different threat categories require distinct identification approaches. In this paper, we design an anomaly detection system for the real-time identification of attacks (adversarial transactions) that target any *code defect in smart contracts* – such as improper access control, reentrancy, undefined behaviors, and even protocol-specific logical bugs. Consider, for example, the attack against the `Euler` protocol [13]: A logic error in the function `DonateToReserve` allowed the attacker to artificially inflate the conversion rate between the borrowed and collateralized assets, leading to the theft of approximately 200 million USD. In §VI, we show that HOUSTON detects this and other similar attacks.

We consider out-of-scope other threats whose root cause is related to malicious administration (i.e., rug-pulls [72]), loss of private keys [14], or loss of administrative control due to a governance takeover [40]. These threats generally involve actions performed by entities (e.g., protocol administrators) with a *high level of privilege* over the protocol. As a result, an anomaly detection system has limited efficacy in addressing such issues, because past actions that were considered "normal" (e.g., updating of an administration key by the protocol administrator) can now turn out to be malicious. Finally, HOUSTON is not designed to detect MEV activities (e.g., *front-running/sandwich attacks*), which are typically attacks that *do not* target code defects in smart contracts.

Technical Challenges. Our core challenge is to design an attack-detection tool for Ethereum transactions that achieves both *generalization* and *precision*, while being *real-time-capable*. (1) Generalizability across attack types with precise detection: Existing detectors often rely on manually crafted, attack-specific rules or on machine-learning features with no clear semantic meaning (e.g., high-dimensional embeddings), limiting both adaptability and interpretability. HOUSTON instead derives vulnerability-agnostic signals from automatically mined invariants and control-flow patterns intrinsic to each protocol. These protocol-specific and semantically meaningful signals produce precise, interpretable alerts, facilitating efficient alert triage and maintaining low false-positive rates over long-term monitoring. (2) Real-time operation on live transaction streams: Detailed data- and control-flow analysis must run within the brief window of block production. HOUSTON achieves this through a lightweight, incremental design that continuously processes traces, detecting exploits in near real-time with bounded computation and memory overhead.

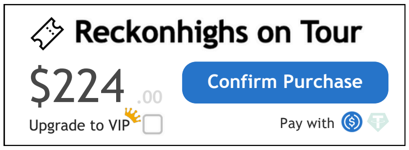

Fig. 1: *TicketMonster*'s Web2 frontend.

```
1  struct Ticket { uint id; bool vip; }
2
3  contract TicketMonster {
4    bool is_open;                        // slot 0x0
5    address admin;                       // slot 0x0
6    uint basePriceInUSD;                 // slot 0x1
7    uint vipTicketSurcharge;             // slot 0x2
8    uint ticketId;                       // slot 0x3
9    mapping(address => mapping(uint => Ticket))
         userTickets;                     // slot 0x4
10   mapping(address => uint) balances;   // slot 0x5
11
12   // ... Bug(A): wrong visibility for function
13   function setVipTicket(address user, uint ticketId,
         bool isvip) public {
14     // ... logic to handle tickets upgrade
15     userTickets[user][ticketId].vip = isvip; }
16
17   function buy(address token, bool isvip) public
         returns (uint) {
18     // ... Bug(B): missing address validation
19     IERC20 _token = IERC20(token);
20     uint totUSD = basePriceInUSD;
21     // apply VIP rate
22     if (isvip) totUSD += vipTicketSurcharge;
23     // call to oracle to get ticketPriceInTokens
24     uint amount = getTokAmount(token, totUSD);
25     require(_token.balanceOf(msg.sender)>=amount, "
         Insufficient balance");
26     require(_token.transferFrom(msg.sender,address(
         this), amount), "Transfer failed");
27     ticketId += 1;
28     Ticket ticket = Ticket(ticketId, false);
29     userTickets[msg.sender][ticket.id] = ticket;
30     setVipTicket(msg.sender, ticket.id, isvip);
31     balances[msg.sender] += 1;
32     return ticketId; }}
```

Fig. 2: *TicketMonster*'s backend handling ticket sales.

IV. Running Example: TicketMonster

To help demonstrate our system, we introduce a hypothetical DeFi protocol that sells event tickets. Figure 2 presents the code snippets of the protocol. *TicketMonster* provides a Web2 user interface (Figure 1) that allows customers to buy event tickets (one per transaction) at a given USD price using stable coins: USDC or USDT. For simplicity, every ticket has a fixed price of $224, with a surcharge of $500 for VIP service. In a typical use case, customers purchase tickets using the website: They would link their crypto wallet (e.g., Metamask [2]), and submit requests to the Web2 website, which, in turn, generates a transaction for the Web3 backend. Importantly, while the transactions generated by the website can be considered safe (i.e., crafted by the developer's code), nothing prevents more sophisticated users from sending requests directly to the smart contract, bypassing any constraints enforced by the Web2 UI. It is easy to note how the Web3 backend contract fails to enforce certain assumptions made by its frontend UI.

Bug (A) The function `setVipTicket` (Line 13) has the wrong visibility: It is supposed to be `internal` rather than `public`. Because of this, attackers could first buy non-VIP (basic price) tickets and then upgrade with no surcharge by directly calling `setVipTicket`.

Bug (B) The second oversight is the absence of validation for the `token` parameter of function `buy` supplied by users (Line 17). An attacker can simply deploy a malicious `ERC20` token contract and use the fake token (instead of `USDC` or `USDT`) for payments. More specifically, when the *TicketMonster*'s contract calls to `balanceOf` (Line 25) and `transferFrom` (Line 26), it interacts with the attacker's token, thus the execution happens according to the attacker's code. This allows attackers to manipulate the return values of these calls and make the purchase succeed without actually paying [75]. Either bug, if exploited, can cause severe financial damage. In the following sections, we show how HOUSTON can identify attack transactions exploiting these bugs.

V. System Design

Our goal is to design an anomaly detector that identifies attack transactions by monitoring contract executions. Instead of relying on predefined attack patterns or entity reputation, the system extracts high-quality, protocol-specific signals from execution traces to produce timely, accurate alerts. Given a DeFi protocol, HOUSTON builds and continuously updates customized anomaly-detection models as part of its ongoing monitoring process. These models are used to evaluate new transactions directed to the protocol. If an anomaly is detected, the system raises an alert. Figure 3 presents an overview of our system and outlines its operational stages. In the following paragraphs, we describe each of these stages.

❶ **Contract Processor**. First, we compile a list of all smart contracts that belong to a protocol. When HOUSTON is deployed, we assume that this list can be easily provided by administrators. For our experiments, however, we developed an approach to determine this list ourselves (see §VI for details). Then, given the set of smart contracts that compose a protocol, we examine the contracts' source code and use the Solidity compiler toolchain to extract relevant metadata, such as their public functions, argument types, and storage layouts [71]. We assume that our system typically has access to the contracts' source code, since HOUSTON users are supposed to be the protocol's administrators (or developers).

❷ **Transactions Processor**. Given the output of the previous step and a transaction that interacted with the protocol (i.e., a transaction that triggered the execution of at least one of the contracts in the protocol), we first collect a detailed execution trace of the transaction, Then, we process this data and extract the necessary features that form the basis for our detection models (see §V-B for details).

❸ **Protocol Interaction Analysis**. The first detection model (*Interaction Model*) leverages the sequences of function calls. Intuitively, a protocol's control flow reflects its core actions and their ordering. However, we are not interested in *all* calls. Instead, we focus on externally initiated calls to the protocol's public interface that either modify persistent storage or trigger key external operations such as token transfers.

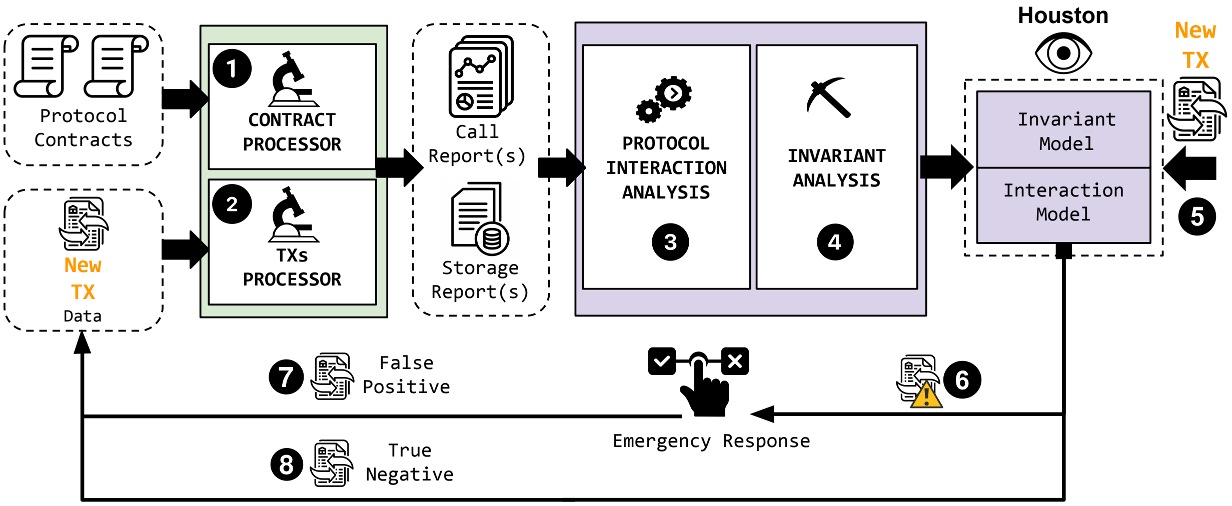

Fig. 3: HOUSTON Overview. The analyses described in ❶-❷-❸-❹ are used to provide data to the models employed by HOUSTON. In ❺, a new transaction is ready to be examined by HOUSTON. In ❻, a transaction is considered to be anomalous, and it is sent to an emergency response system. If the transaction is deemed to be a false positive, it is added to the corpus ❼. HOUSTON will start to generate a new knowledge base automatically. Transactions that do not raise any alert ❽ are also added to the corpus to update HOUSTON's models.

❹ **Invariant Analysis**. For our second model (*Invariant Model*), we focus on the values that are *written* to persistent storage and those passed as arguments to function calls. Specifically, we establish "normal" patterns for these values and their inter-relationships. We believe that the semantics of contracts can be meaningfully captured by the values of their state (storage) variables, their function arguments, as well as the relationship between these variables. To do this, we employ likely-invariant inference techniques [21] to capture important protocol properties automatically.

❺ **HOUSTON Anomaly Detection**. When a new transaction (that targets the protocol) is spotted, HOUSTON (locally) runs the transaction and analyzes the resulting execution trace. Specifically, the two models independently examine various aspects of the transaction trace, including the involved smart contract functions, their argument values, and their validity against the invariants identified in ❹. The outputs from both models are then combined to determine a final verdict. If the transaction is found to be anomalous, we raise an alert.

❻ **HOUSTON Alerts**. Whenever an alert is raised, it can be evaluated by a human analyst or used as a trigger by an automated system to initiate a response. In the event of a confirmed attack (a true positive), administrators can initiate incident response procedures, such as an emergency pause of the protocol [96]. Alternatively, the transaction can be front-run by one of the recently proposed systems that aim to rescue funds [58], [94], [89]. Conversely, if the transaction was erroneously flagged as anomalous (a false positive), it will be incorporated into the transaction corpus ❼. The system will automatically update its models, eliminating the recurrence of similar false positives in subsequent analyses.

❽ **HOUSTON Continuous Updates**. If HOUSTON's models do not raise any alert for a transaction, HOUSTON automatically adds this transaction to its baseline corpus and updates its models with this new information.

A. Contract Processor

Recall that most DeFi protocols consist of multiple smart contracts. When protecting a protocol, our system monitors (and takes as input) all the smart contracts associated with this protocol. These contracts constitute the protocol's footprint on the blockchain. For each contract, we extract its ABI [70], as well as determine its storage layout [71]. These metadata are needed to extract (1) the typed function's arguments from the raw `bytes` of a transaction's `CALLDATA`, and (2) to obtain precise information regarding which storage variables are modified during a contract's execution. To extract these data, we use the off-the-shelf contract's ABI decoder [70], and a custom procedure named *Storage Variables Decoder* ($\mathcal{SVD}$).

Storage Variables Decoder. We design this procedure to extract precise modifications to state variables. More specifically, the $\mathcal{SVD}$ takes as input an arbitrary slot id, the `StorageLayout` of a contract, and a description of the `KECCAK256` operations (i.e., `KeccakInfo`) executed during a transaction (available after tracing a transaction in ❷) and returns fine-grained information about the variables stored in a given slot. This process is necessary because variables in the EVM can be packed together in the same storage slot [71]. Without this distinction, proper annotation and invariant calculation would be impossible. For instance, in *TicketMonster* (Figure 2), the slot id `0x0` contains a Boolean value (i.e., the variable `is_open`) at offset `0` and an address (i.e., `admin`) at offsets `1`→`21`. Computing $\mathcal{SVD}$(0x1,StorageLayout,KeccakInfo) results in $\{v1 = 1, v2 = \texttt{0xADMIN}\}$. Note that we need KeccakInfo to only resolve accesses to dynamic slots (e.g., slots 0x4 and 0x5 in Figure 2). For additional details see Appendix A.

Binary-only Contracts. Currently, if a protocol includes some binary-only contracts, HOUSTON simply avoids calculating the likely invariants in ❹ for those contracts. However, these contracts are still considered for the *Interaction Model* in ❸. We discuss the impact of source code availability in §VII.

B. Transaction Processor

We employ an Ethereum archive node [45] to trace a transaction relevant to a protocol – i.e., one that includes any direct or indirect calls to the protocol's smart contracts. Specifically, we design a custom tracing plugin to obtain information about the following opcodes: SSTORE, CALL, DELEGATECALL, STATICCALL, and KECCAK256.

SSTORE. For SSTORE operations, our custom tracer collects the following information: The slot id being written, the prior value of the storage slot, the new value, the identifier of the function in which this operation happens, the address of the contract executing the opcode (i.e., the code address), and the address of the contract whose storage is being referenced.

CALL. When executing a call (CALL, DELEGATECALL, or STATICCALL), we extract the sender address, the destination address, and the raw bytes of the CALLDATA.

KECCAK256. The KECCAK256 opcode applies the KECCAK256 hash function to its input (the *pre-image*) and produces as output a hash value (the *image*). It is commonly used to compute the address of elements within lists and mappings. For example, consider the mapping balances stored at slot id 5 (Line 10, Figure 2). To retrieve the balance of a user (i.e., purchased tickets) with key USER, the EVM will compute KECCAK256(USER.5) and use the result as an address to access the contract's storage. When tracing a transaction, we record all the pre-images (inputs) for all KECCAK256 opcodes. This information allows us to determine whether writes to different memory locations (i.e., slot IDs represented by different KECCAK256 images) target the same dynamic variable (e.g., a mapping storing users' balances).

For a given transaction, the *Transaction Processor* outputs a *Call Report* and a *Storage Report*. The *Call Report* contains *all* the calls performed by *all* the contracts involved in the transaction execution. The arguments for each function call are annotated by the contract's ABI decoder. The *Storage Report* contains all the available information on SSTORE and KECCAK256 operations, as annotated by the $\mathcal{SVD}$ procedure. In particular, $\mathcal{SVD}$ refines the accessed slot id into the corresponding variables that are modified (in case there are multiple variables packed into a slot). As a result, the *Storage Report* contains detailed information on every variable modified in the storage of any contract in the protocol. For instance, in *TicketMonster* (§IV), we would annotate a Ticket purchase with its corresponding id and vip flag.

C. Protocol Interaction Analysis

The sequence of function calls executed by the protocol's smart contracts (throughout a transaction) provides valuable insights into a protocol's behavior. Intuitively, if a sequence of function calls deviates from typical and common patterns, it is an outlier, and, often, these outliers are attack attempts [30]. However, the entire call sequence of executed functions (within a transaction) can be very long (e.g., hundreds of nested function calls), may include activities unrelated to a monitored protocol, or may contain uninteresting protocol actions (i.e., a call to a function that simply returns the balance of a user). This can add noise to a model and lead to false positives. Thus, unlike prior work that inspects all function calls, we focus *only* on "critical" function calls that are relevant to the monitored protocol. We name the list of critical function calls *Protocol Interaction*. Our *Protocol Interaction* analysis unfolds in four steps:

(1) Call Direction Identification. The analysis first categorizes all calls (that are part of a transaction trace) according to the "direction" of the call. Specifically, we only keep *incoming calls*, that is, calls from a smart contract (or externally owned account) whose address is *not part* of the protocol, to a smart contract *within* the protocol. By retaining only incoming calls, we construct protocol interactions that account for actions initiated by entities external to the protocol. This significantly diminishes the noise caused by outgoing and internal calls (see our ablation study in Appendix C).

(2) Critical Call Identification. Based on the execution tree provided by the *Call Report*, and the storage writes reported in the *Storage Report*, this analysis marks any incoming function call as *critical* if either the called function itself (or any other function within its call tree), (1) triggers storage changes (performs an SSTORE operation) in any of the smart contracts belonging to the DeFi protocol or (2) initiates the execution of a function matching a predefined list of identifiers. Currently, this list includes four standard ERC20 token [24] functions: transfer, transferFrom, safeTransferFrom, and approve. Intuitively, these functions are important because they capture token transfer and approval mechanisms, thereby capturing the movements of digital assets.

(3) Direct ERC20 Token Operation Filtering. Many DeFi protocols include one or more ERC20 token contracts. External users (and contracts) can directly interact with these token contracts, performing operations (e.g., transfer) without involving the rest of the protocol's logic. These operations tend to occur in large volumes and generally do not correlate to potential attacks but rather add substantial noise to the model. Therefore, HOUSTON filters out all *incoming* ERC20 calls from the protocol interaction pattern. Importantly, we only exclude *incoming* ERC20 operations (e.g., approve in Figure 4). If a call to a non-ERC20 function triggers downstream token movements (e.g., a call to transfer), HOUSTON still considers it as a part of the protocol interaction, e.g., in Figure 4, the *incoming* call to function pay triggers an ERC20 transfer, thus, it is part of the final fingerprint.

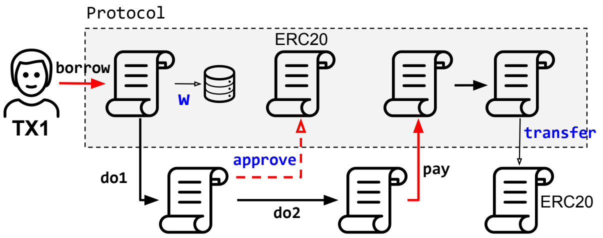

Fig. 4: Protocol Interaction Analysis. The protocol's smart contracts are in the gray area. Function `borrow` triggers a storage write in one of the protocols of the contract. Function `approve` is filtered out. The function `pay` leads to an `ERC20 transfer`. The fingerprint of `TX1` is `[borrow, pay]`, but we save function selectors instead of their names.

(4) Interaction Pattern Identification. We use the sequence of critical function calls as a *fingerprint* of the transaction. Specifically, from the *Call Reports* of historical transactions, the *Protocol Interaction Analysis* builds a database of all observed interaction fingerprints (later used by the *Interaction Model*). For example, in *TicketMonster* (§IV), as benign users buy tickets via the Web2 UI, the protocol interaction database would contain only one fingerprint: `[buy]`. The reason is that neither the call to the price oracle (Line 24) nor the call to function `setVipTicket` (Line 30) is an *incoming call* (i.e., a call coming from outside the protocol and directed to it), thus, they are not considered in the final fingerprints. To better illustrate our algorithm, Figure 4 shows a more complete example of how we reduce a complex sequence of calls (in and out of a protocol) to the signature `[borrow,pay]`. By focusing on critical functions, we significantly reduce the complexity of call sequences while preserving the relevant information regarding the protocol's function calls. For example, in the `Euler` protocol attack, the attack transaction made 150 calls, whereas the *Protocol Interaction* representation of the attack comprises only 8 calls. These 8 calls outline the exact steps of the attack as security experts described in their writeups [61]. In our experiments (§VI), HOUSTON successfully detects the attack transaction with the help of these fingerprints. Moreover, we show that the focus on critical functions (rather than modeling the entire function call trace) is critical to the accuracy of this model (Appendix C).

D. Invariant Analysis

Program Invariants. Invariants represent properties within a program's code that always hold at a program point or a set of program points [21]. Commonly, identifying program invariants requires static analysis techniques, such as symbolic execution and abstract interpretation [66]. However, those techniques suffer from scalability limitations when used to analyze complex contracts. Moreover, for our purposes, we are not interested in proving that certain program invariants always hold. Instead, our focus is on identifying relationships between variables that consistently hold in incoming transactions, reflecting the intended behavior of a smart contract's execution. These relationships, when violated, may indicate an attack. Hence, for HOUSTON, we collect *likely invariants* [21].

Likely invariants are properties that are dynamically inferred from values observed during multiple program executions. For instance: in *TicketMonster*, the argument `token` has been historically observed to be equal to `0xUSDC` or `0xUSDT` (these are the only two currencies allowed for payment by the Web2 UI, Figure 1). This dynamic inference approach generally produces relevant relationships, but it comes at the cost of soundness (that is, a likely invariant may not hold true for all executions). Nevertheless, given enough observations of a likely invariant, the confidence of the observed property increases [22]. The goal of the program invariant analysis is to automatically identify likely invariants at the function boundaries of a smart contract. Specifically, we infer relationships between a function's inputs (its arguments and initial storage state) and its outputs (represented by the resulting storage state). Unlike return values alone, storage updates capture meaningful behavioral effects because persistent state changes influence both the current execution and future interactions.

Likely-Invariant for Attack Detection. Likely-invariants naturally arise from the execution of DeFi protocols. Our core intuition is that likely-invariants can be used to identify attacks against a protocol, at the cost of tolerating a small percentage of false positives (i.e., "benign anomalies"). Intuitively, since attack transactions are intrinsically abnormal (i.e., they exploit edge cases in the protocol), they are likely to breach one of these likely-invariants, and, therefore can be observed.

Likely-Invariants Class Selection. The selection of likely-invariant classes requires considering a wide range of possibilities. Intuitively, mining *all* possible unary and binary relationships among *all* the variables in a smart contract would rapidly make the problem intractable (e.g., the Daikon [21] engine offers more than 200 invariant classes). To reduce the number of invariant classes, we initially experimented with a broad range of them on a pilot dataset of 16 DeFi incidents covering diverse attack types. While experimenting with different invariants on our pilot dataset, we observed several practical challenges: Some invariant types were overly sensitive and triggered frequent false positives (e.g., the `UpperBound` invariant, which alerts whenever a variable exceeds its historical maximum); others (e.g., ternary invariants) led to a combinatorial explosion in run-time or lacked clear semantic interpretation. We iteratively filtered out such invariant classes and focused on those that both: (1) showed violations during real attacks, and (2) had clear, human-interpretable semantics. This trial-and-error process led us to the following four simple invariant classes used in the final system:

(C1) For individual integer variables, the analysis checks whether a variable is *never* zero. An unexpected zero may signal abnormal behavior in arithmetic operations.

(C2) For individual address and bytes variables, the analysis determines if the address or bytes variable always equals a specific value, or if it is drawn from a set of possible values.

(C3) For relationships between two integer variables, the analysis checks if any of the ordering relations ($=,\geq,\leq$) always hold. For example, an invariant may state that the amount of redeemable tokens is *always* less than the user's balance.

(C4) For relationships between two strings, addresses, or bytes, we evaluate whether the two variables are always the same. This invariant type can help when, for example, the developers assume that two addresses should always be identical to ensure that a transaction originates from the same source, but fail to enforce the assumption.

Selective Mining for Binary Likely Invariants. The attentive reader may have noticed that our method for generating likely invariants over the contracts' variables adopts a "brute-force" strategy. Specifically, HOUSTON checks all possible relationships or properties (of types **C1–4**) over variables and variable pairs as they are observed during the processing of a protocol's transactions. While this approach is acceptable for unary invariants, it quickly leads to the generation of numerous spurious binary invariants that lack any real semantic meaning (e.g., timeStamp < reserve). Although many of these invariants are unlikely to ever be violated, some may introduce false positives. To address this, we precompute all variable pairs in each contract that the *Invariant Model* might compare and use an LLM (GPT-4.1 [53]) to check if each comparison is semantically meaningful in the context of the contract code (see Appendix H for the prompts). During monitoring, the *Invariant Model* only mines (and checks) over the variable pairs deemed semantically meaningful. We quantify the effectiveness of this selection step in §VI-C.

Likely-Invariants Inference. To identify likely invariants, we combine the information from *Call Report*s and *Storage Report*s (generated in ❷) to create data traces of function arguments and storage variables at the entrance and exit of all function calls. The *Invariant Analysis* then determines, for each invariant category of interest, whether an invariant over a single variable, or between two (semantically comparable) variables, consistently holds across *all* executions at a specific program point. If this consistency is verified, we record the invariant and the number of transactions supporting it (as a metric of confidence) for future reference. Importantly, our current invariant inference algorithm is *not* designed to extract sophisticated invariants of the kind that a developer might manually specify (e.g., collateralValue × collateralFactor ÷ debtValue ≥ 1, which was violated in the Euler incident). Instead, its goal is to mine a large number of *simpler* candidate likely invariants. While some may be quickly violated (and thus deemed unlikely), others capture meaningful properties with broad coverage, emerging as valuable indicators for detecting attacks, either directly or through the side effects of malicious transactions. Our invariant extraction algorithm can be easily extended with more advanced systems [46] if required by the protocol.

E. HOUSTON *Anomaly Detection*

For every new transaction, HOUSTON creates an execution trace and consults its models to identify anomalies.

Interaction Model. The *Interaction Model* computes the transaction's fingerprint as described in §V-C and compares it against those in the database. The model reports an anomaly if the fingerprint has not been seen before. For example, if an attacker tries to exploit *TicketMonster* using Bug (A) (Figure 2, Line 13) by first calling the buy function and then, in another transaction, calling setVipTicket to upgrade their ticket, they would generate an unseen fingerprint [setVipTicket].

Invariant Model. The *Invariant Model* observes all protocol transactions and computes the likely invariants. Then, it uses such invariants to detect anomalies (invariant violations) in new transactions. In the ideal case, where the monitored application is stable and has seen significant usage, a transaction should be flagged as anomalous if *any* of the computed invariants is violated. For instance, in our *TicketMonster* example, an attacker attempting to exploit Bug (B) by providing a token address other than 0xUSDC or 0xUSDT would violate the invariant $\texttt{token} \in \{\textit{0xUSDC, 0xUSDT}\}$, which is an invariant of category **C2** as described in §V-D. For newly deployed contracts or functions that see limited use, the supporting observations for an invariant may be minimal. This scarcity of data can make it challenging to determine whether an inferred likely invariant accurately reflects the core semantics of the contract or whether the invariant is merely incidental due to insufficient transaction diversity, suggesting that alerts on its violation may be groundless. To address this, the *Invariant Model* enforces a waiting period for each likely invariant, allowing more user interactions with the protocol. During this period, any violation will lead HOUSTON to discard the invariant without triggering an alert. The waiting period is defined by two hyperparameters: the minimum transaction count supporting the invariant (N) and the minimum contract age (T). A violation is within the waiting period only if *both* conditions are met. In this study, we set $N = 10$ and $T = 12$ hours, but our experiments show that detection outcomes are stable across various settings (see Appendix D for more details). The final decision is simple: If *at least one* of the two models report(s) an anomaly, HOUSTON triggers an alarm.

F. HOUSTON *Continuous Updates*

HOUSTON continuously incorporates new interaction fingerprints and updates invariants, keeping its behavioral models aligned with the protocol as it evolves. This incremental learning is essential to maintaining a low false-positive rate.

***Interaction Model* Update.** Whenever a new protocol interaction is observed, HOUSTON raises a warning and then simply adds the interaction fingerprint to its database.

***Invariant Model* Update.** This involves three main operations: mining new invariants, discarding any that have been violated, and increasing the confidence of existing invariants by incorporating additional observations. For example, assume the invariant $\texttt{token} \in \{\textit{0xUSDC}\}$ has been established for *TicketMonster*. If a new transaction uses 0xUSDT as token (Figure 2, Line 18), HOUSTON will raise a warning, discard the previous invariant, and establish the new invariant: $\texttt{token} \in \{\textit{0xUSDC, 0xUSDT}\}$. To enable continuous updates of the *Invariant Model*, we implemented our own invariant mining engine inspired by Daikon, a state-of-the-art system for inferring likely invariants [21]. Our invariant miner adds support for incremental invariant inference, which enables HOUSTON to quickly incorporate new information into its models.

VI. EVALUATION

A. DeFi Attacks Dataset

To evaluate HOUSTON, we compiled 115 DeFi protocol incidents into a comprehensive benchmark, drawing from both previous work [30], [96] and public reports of security incidents [62], [76]. We curated our dataset to be as comprehensive as possible by including all incidents within the defined scope. Only those incidents that fell outside this scope (as outlined in §III) or occurred on non-Ethereum chains were excluded. Our dataset includes protocols of varying application types and exploits against a diverse set of vulnerabilities. Note that each incident corresponds to a distinct protocol. For incidents with multiple attack transactions, we label the *earliest* transaction as the ground truth, as detecting this initial action allows administrators to respond swiftly, for instance, by pausing the protocol to prevent further damage. Notably, this approach is more challenging than merely identifying any one of the attack transactions. For each attack, we report the address of the contract being attacked, the block and date of the incidents, and the hash of the first attack transaction. Appendix B, Table II, summarizes the DeFi protocols in our dataset.

Identify Smart Contracts of a Protocol. In §V, we described how our system takes as input the addresses of the smart contracts belonging to a DeFi protocol and then uses incoming transactions to build its models. In practice, protocol administrators can (easily) identify the involved contracts. For our evaluation, however, this information is not readily available. We therefore approximate protocol membership as follows. We first identify the deployer of the attacked (victim) contract, i.e., the account that created it. We then include all contracts directly deployed by this account as part of the protocol. Since contracts may also be created through internal transactions, we further inspect the deployer's internal transactions and include any contracts created in this way.

B. Experimental Setup

We evaluate our system on a server with dual Intel Xeon Gold 6330 processors, 512 GB of RAM, and 20 TB of SSD storage. To evaluate HOUSTON's capabilities, we design two experiments: the *Historical Evaluation* and the *Live Performance Evaluation*. The goal of the former is to demonstrate the system's high precision and recall across 115 historical attacks, while the latter aims to showcase its effectiveness in real-time scenarios when monitoring 20 protocols in parallel.

C. HOUSTON *Historical Evaluation*

Given a protocol, we first collect all past transactions that interacted with any of its contracts, up to the point of the first known attack (as per our dataset VI-A). HOUSTON then analyzes these transactions in chronological order to identify anomalies. We record whenever an alert is produced, and we stop our evaluation after processing the reference attack transaction. In this evaluation, we consider any alert that does not correspond to the reference attack transaction to be a false positive. While this approach may be imprecise—particularly if there are unknown attacks preceding the reference incident—it provides an upper bound on the number of false positives generated by HOUSTON. It is also worth noting that while an alert from HOUSTON might be considered a false positive, the flagged transaction may still exhibit anomalous features that can offer valuable insights, e.g., a new behavior used in the protocol, or a new contract appeared in the protocol that is interacting with it in new ways. Finally, when evaluating the attack transaction, if HOUSTON flags the transaction as anomalous, we consider it a true positive; otherwise, we consider it a false negative. Our results show that HOUSTON is effective in detecting attacks for a diverse set of protocols. Specifically, out of 115 DeFi protocol incidents in our dataset, HOUSTON detected 109 attacks (∼94.8%). HOUSTON also exhibited a low false positive rate of 0.16%. We present a per-protocol results overview in Table VI in Appendix I.

True Positives. In contrast to detection systems based on neural networks [30], the alerts produced by HOUSTON have a high level of explainability. That is, they directly show which calls are anomalous or which invariants have been violated. This makes it much easier for human analysts to determine the root cause of an attack and to discard potential false positives. This level of explainability enables us to systematically examine how HOUSTON 's detected anomalies align with the actual exploit mechanisms. Specifically, we analyzed all 109 detected incidents and compared our alerts with the corresponding public postmortem reports. For each incident, we examined alerts from both models and used the more conclusive one as its representative signal. Among these incidents, HOUSTON produces *causal* detections in 57 (52%) cases, where the invariant violation or interaction pattern directly pinpoints the root cause of the vulnerability. In these situations, the abnormal behavior exposed by HOUSTON is so fundamental to the exploit that carrying out the attack without triggering such an alert would have been impossible. Another 43 (39%) incidents fall into the *indicative* category, where the detected violation reflects a consequence of the attacker rather than the immediate exploitation of the bug. These alerts often capture secondary effects that reveal the attacker's intent or correspond to side effects of the exploit. While such signals might be avoided by a highly sophisticated attacker, it depends on the specific circumstances or goals of the attack, and doing so would significantly increase its difficulty. Finally, 9 (8%) incidents are *incidental* detections, where the triggered signal is unrelated to the exploit and uninformative for triage. Most of these cases are in protocols exploited early in their deployment, and the alerts are raised because the transaction paths are novel.

Below, we discuss two interesting true positives that our system detected (refer to Appendix G for more examples).

(1) *RevestFinance.* This attack was detected by both of HOUSTON's models. Specifically, the invariant violation points to a mismatch between the two storage variables `fnftID`

and fnft_created, which are supposed to be equal when entering the function mint (**C3**, §V-D). This mismatch was caused by the exploitation of a reentrancy vulnerability, and the attack could not have succeeded without triggering this condition (*causal*). For the *Interaction Model*, no previous transaction ever called mintAddressLock twice in its protocol interaction (hinting at possible reentrancy) and thus triggered a warning (*indicative*).

(2) *Euler.* Both HOUSTON's models identified the attack. Specifically, the *Interaction Model* reported a new protocol interaction where a call to donateToReserves preceded a call to liquidate, which is essential for the exploit to succeed (*causal*). At the same time, the invariant model reported a violation during the execution of liquidate. In all prior executions, deferLiquidityStatus was *never* zero for liquidate (**C1**, §V-D). This violation arose because the attacker deliberately skipped deferLiquidityChecks, which is normally invoked by all legitimate users. Although the exploit could still succeed even with this check in place, omitting it reflects a distinct behavioral deviation – an attacker's willingness to bypass standard safety routines. Therefore, we classify *Invariant Model*'s detection as *indicative*.

False Negatives. HOUSTON failed to detect an attack in only 6 incidents out of 115 (5.2%). After manual investigation, we observed that four incidents would require mining additional types of invariants. For instance, for the nomad incident [15], it would be necessary to infer invariants on values read from the storage (currently, we focus only on storage writes, as discussed in §V-B). Of the remaining two cases: azukidao was exploited too soon (4 days) after deployment, and, therefore, HOUSTON simply did not have enough data to learn anything meaningful; snood's exploit directly targeted a buggy re-implementation of a transferFrom in the protocol, and since incoming ERC20 calls are filtered to reduce false positives (as explained in §V-C), this attack was not detected.

False Positives. Overall, HOUSTON had 13,827 false positives out of 8,586,421 transactions evaluated across all 115 protocols over their entire history (false-positive rate 0.16%, 0.4 false positive per protocol per day on average). As is shown in Figure 5, 64 (55.7%) protocols have $\leq 1\%$ false positive rate; 92 (80.0%) protocols have $\leq 3\%$ false positive rate. Table VI in Appendix I shows the detailed false-positive numbers (and rates) for each protocol. We investigated these false positives and identified two main reasons for these errors:

(1) *Historical Data Availability.* When there are too few observations (or no observations in the case of new contracts), HOUSTON has very little information to learn from and will suffer from false positives during a short initialization period. Luckily, as transactions accumulate, HOUSTON's invariants and interaction fingerprints become more reliable, and the false positive rate drops quickly. In Figure 6, we show the correlation between the number of transactions available for a protocol and the false positive rate. Notably, *all* of the 23 protocols with a false positive rate exceeding 3% had fewer than 800 total transactions. In practice, developers can mitigate this cold-start effect by bootstrapping HOUSTON with their own test cases or simulation traces prior to deployment. These synthetic transactions provide initial behavioral diversity for the model to learn from, allowing it to establish preliminary invariants and interaction patterns and, thereby, significantly reducing false positives during the early monitoring phase.

(2) *Likely-Invariant Quality.* In §V-D, we described how we used an LLM to selectively mine binary invariants to avoid nonsensical pairs. We found that this selection step reduced the number of false positive alerts in the *Invariant Model* by 27.2% (from 10,579 to $7,702$), while preserving the model's detection capability (true positives remain unchanged). Because our filtering is purposefully conservative (see our prompts in Appendix H), some spurious relationships might persist. We manually analyzed a random sample of 100 mined invariants from the *Invariant Model*. Of these, 84 were deemed meaningful: we could imagine plausible scenarios in which violations of these invariants can be indicative of abnormal executions (even if such violations may never occur in practice). Among the remaining 16 that we found to be spurious, two were unary invariants that asserted unjustified restrictions on free parameters. For instance, one such invariant incorrectly assumed that the spender parameter in the allowance function must belong to a small, fixed set of addresses. The remaining 14 were binary invariants involving variable pairs that are not semantically comparable. This could be further resolved by utilizing a better reasoning model at the cost of a more expensive pre-filtering step (the process cost approximately $100 of LLM credits across the entire dataset).

(3) *Operation Repetition and Permutation.* The *Interaction Model* drastically reduces the call trace of a transaction to a few, meaningful operations. However, the repetition (and permutation) of such operations can lead to new interaction fingerprints that cause false alarms. For instance, if a user performs five deposits and one withdraw against a monitored protocol, depending on where the withdraw is placed among the five deposits, the transaction may generate six unique fingerprints. Regardless, these false alarms appear less frequently as transactions accumulate. To mitigate this, one may associate each operation with the underlying logical entity it acts upon (e.g., a stake or lien). This entity-level grouping effectively partitions long, flattened parallel sequences into independent sub-sequences, thereby preventing the combinatorial explosion of possible interaction patterns.

Given the statistics and root-cause patterns above, it is useful to consider the operational impact of false alerts. Although HOUSTON produces false alerts on the whole historical dataset, this does not undermine its practicality. Most false positives occur during the initial (learning) phase, after which the rate stabilizes at very low levels, with mature protocols typically seeing fewer than one non-exploit alert per day (see Figure 6, Table VI). Because alerts are explainable and closely aligned with actual exploit mechanisms, HOUSTON remains effective despite the presence of false positives. To further evaluate the false positive rate of HOUSTON on mature, high-quality protocols without known history of exploitation, we collected all transactions up to the time of experiment

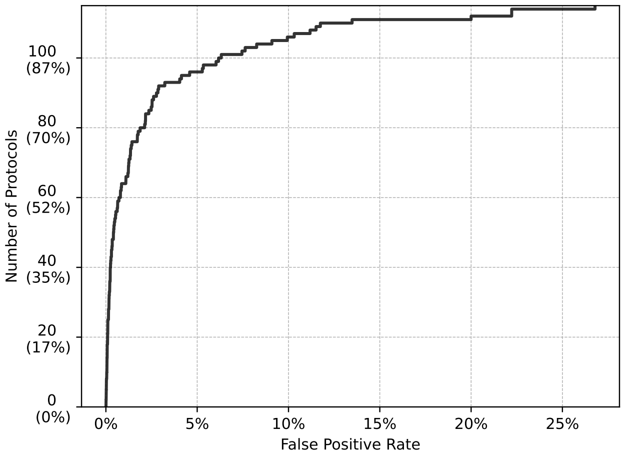

Fig. 5: Cumulative distribution of false positive rates.

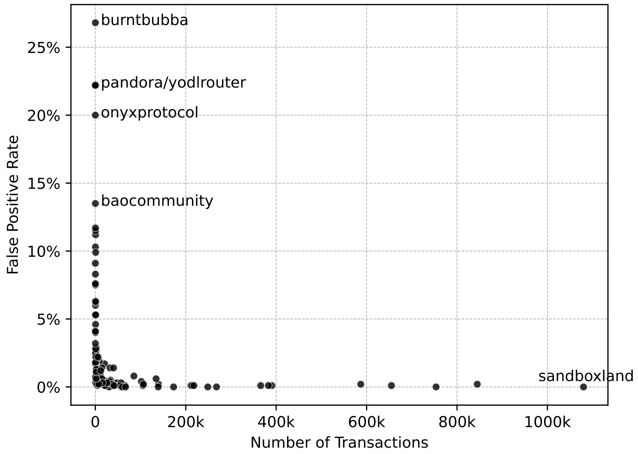

Fig. 6: Overview of the correlation of false positive rates with the total number of transactions available per protocol. Note that protocols with high false-positive rates are evaluated with an exceptionally low number of transactions. See Appendix I Table VI for detailed per-protocol results.

(block 20400000, totaling over 13 million) of an additional set of popular Ethereum DeFi protocols with TVL (Total Value Locked) greater than $500M [19] (see Appendix F for the complete list). Then, we performed the same experiment as the one for the DeFi attacks dataset. For this experiment, we reported an aggregate false-positive rate of 0.08%, which is significantly lower than that for the DeFi attacks dataset. This supports our observation that, for well-used protocols, false-positive rates drop significantly, indicating that our system is effective in real-world settings, where accuracy is essential. Finally, to reduce false positives, one could also initialize Houston's models with existing benign transactions.

D. Comparison with Existing Systems

APE. APE is a system that automatically synthesizes adversarial smart contracts to imitate (and front-run) profitable transactions in real-time [58]. Although Houston and APE pursue fundamentally different objectives (anomaly detection vs. profitability detection), a comparison between the two is still meaningful, as Qin et al. suggest that APE can also be leveraged as a defensive mechanism. For this comparison, we were unable to run APE directly (the system is not open-sourced); however, the authors provided us with the list of transactions that APE historically replayed on the Ethereum mainnet. This transaction set spans from block 12936340 to block 15253305, overlapping with our dataset. The overlap includes 2.4M transactions, among which there are 20 attacks. Out of these 20 attacks, APE successfully countered 4, while Houston flagged 19 (both systems share a false negative). Within the same reference time frame, APE exhibited a false positive rate of 0.15%, compared to 0.10% for Houston. Theoretically, Houston can enhance APE's attack capabilities when both target the same DeFi protocol. Upon receiving a transaction, APE synthesizes a frontrun, while Houston checks if the activity is malicious. If confirmed, APE can promptly deploy the crafted transaction.

BlockGPT. Gai et al. proposed BlockGPT [30], a generic approach for DeFi anomaly detection based on neural networks. At the time of writing, the authors have not yet made their system available as open source. Thus, we chose to compare our results based on the protocol attacks reported in their paper in Figures 5, 10, and 11. Also, the authors did not disclose the exact transaction hashes that BlockGPT marked as attacks. Thus, we must assume that BlockGPT identified the same transactions as our investigation. While we understand this can create discrepancies in the comparison results, this is our best attempt to make a fair comparison. To compare with BlockGPT, we selected the 28 attacks that are both in scope for Houston (i.e., on Ethereum) and present in BlockGPT's dataset. Overall, our system detects 27 attacks across these 28 protocols, whereas BlockGPT detects only 15. Our only false negative is 88mph. Unfortunately, it is not clear why BlockGPT would detect an anomaly against 88mph because (1), the system naturally suffers from limited explainability, and (2) we have no information regarding the transaction hash labeled as an attack by the authors. Table VI shows a detailed comparison between the detection results for Houston and BlockGPT. Since we do not have access to BlockGPT, we cannot run it directly on our dataset to determine its false positives. Instead, we look at the reported results in Figure 4 of the paper. Overall, BlockGPT achieves a false positive rate of less than 10% for 58% of the protocols in its scope. As a comparison, Houston has a false-positive rate of less than 10% for 92.2% of the protocols. Notably, Houston achieves a false positive rate of 1% or less for 55.7% of the protocols.

TxSpector. TxSpector [91] is a logic-driven framework that detects attacks by analyzing execution traces. It relies on user-defined control and data dependencies rules tailored to specific attack patterns (the original paper covered, for example, re-entrancy, unchecked call, and suicidal vulnerabilities). As a consequence, the range of attack types TxSpector can detect is limited. When applied to our dataset, it identified only 25 out of 115 attack transactions (21.7%), whereas Houston achieved a true-positive rate of 94.8%. Moreover, not all vulnerability types can easily fit into the framework, making it impractical to rely solely on writing additional custom rules to achieve more generic detection coverage.

DefiRanger. The system targets price manipulation attacks on DeFi applications [88] by constructing and analyzing a cash flow tree. It successfully identifies attacks on several protocols in our dataset, as detailed in Section 6.2, Table 5 of their paper. Of the 9 reported attacks, we excluded `Plouto` (Binance chain), `DRC`, and `MET` (missing transactions' hashes). Our evaluation shows that Houston detected all 6 remaining attacks and flagged the correct transaction as anomalous. Additionally, our dataset includes 23 other price manipulation attacks not included in `DeFiRanger`'s dataset, which it could plausibly detect (though this is unverified due to lack of open source). The remaining 86 attacks (74.8%) in our dataset involve other vulnerability classes and would likely be false negatives for `DeFiRanger` (out of scope).

Transaction Length Baseline. It seems to be common wisdom in the blockchain community that it is sufficient to look for abnormally long transactions to detect attacks. While this might be true for some exploits, we argue that transaction length is generally a weak signal. To demonstrate this, we evaluate the transaction-length-based approach by using a simple heuristic that just measures the length of a transaction (in terms of the number of function calls performed) and compares it to the maximum length observed up to that point. We raise an alert whenever a new transaction is longer than any observed before. As expected, this basic approach detects only 46 attacks (40%) out of the total 115.

E. Houston *Live Performance Evaluation*

It is essential for anomaly detection systems in DeFi to keep pace with block production. Specifically, in the context of Ethereum, a system must be able to evaluate all the transactions committed to the latest block in less than 12 seconds (*block-time*). Moreover, to offer attack-prevention capabilities (i.e., identifying attack transactions in the mempool before they are committed to a block), a system must also manage the influx of transactions sent to the mempool. Therefore, it is necessary to test Houston in a real-life setting to demonstrate how it can keep up with both the pace of the live production of blocks *and* the transactions sent to the mempool. To evaluate Houston as a real-time anomaly detector, we deploy it live against 20 randomly selected protocols (from our dataset) that are operational at the time of this experiment (protocols marked with $\star$ in Table VI, Appendix I). We deployed Houston to monitor new incoming transactions as discussed in §V. Houston analyzed *all* mainnet transactions from April 9th to April 28th, 2024 (20 days). This includes transactions in the mempool *and* committed blocks.

Transactions Filtering. Houston observed a total of 42.6M transactions.We discarded 50.2% of them either because they perform simple ETH transfers (no smart contract involvement), or because they directly invoke basic functions of well-established token contracts (e.g., `USDC transfer`). This step took a few nanoseconds per transaction, which is negligible.

Tracing Performance. To check if a transaction calls the monitored protocols, Houston must replay it to collect a trace. This is because simply checking for direct invocations is insufficient; internal transactions also need to be considered. On average, we observed $\sim$13 transactions per second. Given the average tracing time of 3 ms per transaction, a single *tracing worker* can handle this load in 39 ms when monitoring all 20 protocols simultaneously. Note that a *tracing worker* only requires $\sim$150MB of RAM and 25% of a CPU core.

Processing Performance. If a transaction involves *any* of the contracts in a monitored protocol, Houston needs to extract the *Call Report* and the *Storage Report* ❷. Overall, during live experiments, we observed a total of 95.6K transactions directed to the monitored protocols. On average, the number of transactions for which this step is required is about one transaction per block. Given our average processing time of 180ms, this load can be handled by one *processing worker* equipped with $\sim$1.5GB of RAM and 1 CPU core.

Detection Performance. After processing, Houston performs the following two steps: (1) Houston consults its models to make a detection decision ❺, and (2), it updates the models with the new transaction's information ❼/❽. Together, these two operations take, on average, $\sim$28ms per transaction. The resources required by a *detection worker* are 2GB of RAM and 2 cores. Each monitored protocol needs at least one dedicated *detection worker*.

Worst Case Performance. The average load for Houston to monitor 20 protocols is light and could be handled by one *tracing worker*, one *processing worker*, and 20 *detection workers*. Of course, our system cannot just be built for the average case. We also need to take into account bursts of transactions. During our live monitoring, we observed the following peak instances: A maximum of 362.9 transactions/second that needed to be traced, as well as 2.2 transactions/second that required further processing. Moreover, we observed individual worst-case processing times of 337ms for the *tracing worker*, 2.9s for the *processing worker*, and 810ms for the *detection worker*. See Appendix E for a summary of average and worst-case performance (Table III) and transaction load (Table IV). Overall, combining all worst-case instances, Houston would require 121 tracers, 8 processing workers, and 40 detection workers (2 per protocol). Given these requirements, Houston needs $\sim$110GB of memory and 120 cores to keep pace with real-world Ethereum mainnet traffic. These requirements can be matched by a single modern server, such as the one we used in our evaluation, where we never encountered all worst-case instances simultaneously. Importantly, we think this experiment provides evidence of feasible real-world deployment. Should Houston be adopted in an industrial setting, there are clear engineering venues for achieving faster tracing and detection times (e.g., using reth [55] instead of Erigon [45], and re-implementing Houston in Rust [65]).

Live Setup. Our live evaluation used 8 tracers, 4 processing workers, and 20 detection workers (one per protocol). The system showed no latency or resource issues: The transaction queues remained empty, memory peaked at ∼80GB, and CPU load averaged 8.0 across 112 cores.

Live Monitoring Warnings. During the live evaluation, HOUSTON kept low false positive rates. Overall, HOUSTON raised a total of 75 warnings (0.08% of the 95.6K transactions directed to monitored protocols). For 9 of the 20 monitored protocols, HOUSTON never raised an alert. For 10 of them, HOUSTON raised fewer than 10 alerts throughout the 20-day period. The only exception is `conicfinance`, which reported 24 warnings over 901 transactions (2.7%). The reason for this is that a newly introduced `ConicPool` address triggered multiple invariant violations at various program points.

Additional Monitoring. We conducted an additional three-week live experiment (October 27th to November 16th, 2025) on the same 20 protocols. Across 108.5K related transactions, HOUSTON generated 71 alerts (0.07%), consistent with the false-positive rate in the previous period. Manual analysis confirmed that none were attacks, and no external sources reported incidents during this period.

F. HOUSTON *for other EVM-compatible chains*

To further validate the generality of our approach, we ported HOUSTON to another major EVM-compatible blockchain: *Binance Smart Chain* (BSC [6]). In general, three key components are required to adapt our system to a new EVM-compatible chain: (1) a **native transaction tracer**, (2) a **blockchain index database**, and (3) a **contract analysis tool** capable of fetching and inspecting smart contracts to extract their *storage layouts* and *ABIs*. To this end, we ported our Erigon plugin to bsc-erigon [52], we used Dune Analytics [20] to fetch the protocol's boundaries and related transactions, and we adapted the toolchain in Figure 3, Step ❶, to work with BSC. For this experiment, we consider the incidents listed on *DeFiHackLab* [76] that occurred on BSC between May and October 2025 (20 incidents over six months). Among these, we discard seven incidents because the vulnerable contracts are available only in binary form, and an additional one due to insufficient information about the vulnerable contract. HOUSTON successfully identifies attacks in 11 of the remaining 12 incidents, with a low false positive rate of 0.19%. The only undetected case (`pdz`) does not represent a genuine detection failure: HOUSTON instead flags an earlier transaction (`0x88f8741d`) that occurred 202 days before the labeled incident and exhibited a highly similar price-manipulation pattern. Notably, this earlier attack was never publicly reported; the protocol's maintainers appear to have been unaware of it and continued normal operation.

VII. Discussion and Limitations

Source Code Availability. The precision of HOUSTON's models depends on the availability of contract ABIs and storage layouts. Although partial information can be inferred through binary-only analysis [33], [64], [35], accuracy generally degrades. Accordingly, we assume developers using HOUSTON have access to their own source code. However, third parties analyzing unverified contracts must rely on decompilers to recover ABIs and storage layouts. Notably, recent advances [25] leveraging LLMs have achieved up to 49.8% success in recompiling binary-only contracts, enabling the extraction of ABIs and storage layouts. Practically, our contract processor (Step ❶ in Figure 3) would need to be modified to automatically decompile and recompile unverified contracts to produce the required artifacts. Once these artifacts are obtained, the rest of HOUSTON's pipeline is unchanged.

Private Transactions. HOUSTON can prevent or detect abnormal transactions, whether before mining or after inclusion in a block. Clearly, prevention requires the ability to observe transactions in the mempool. However, under certain circumstances, this may not be feasible. For example, users might utilize private relayers (e.g., Flashbots [1]) or verifiers could insert transactions themselves before proposing a block. In such scenarios, a transaction will not be visible in the mempool, preventing HOUSTON from facilitating preemptive measures. This limitation could be mitigated if private relayers and/or verifiers adopt HOUSTON. Notably, the private industry is currently deploying systems similar to HOUSTON [28], [37], [39], [57], as a defense-in-depth strategy, demonstrating the value of such a system. These systems differentiate L2 services, which might censor attack transactions at the sequencer level, and support L1 blockchains in implementing protective measures when direct censorship is impractical.

Dynamic Adversary (Poisoning). Poisoning HOUSTON is possible in principle, but not without difficulty. Consider a contract where functions `A` and `B` can change the `openMarket` flag. While both functions should be protected by `onlyAdmin`, only `A` enforces it. An attacker observes that invoking `B` can close the market and impact the protocol, but the direct invocations of `B` would trigger an alert. A feasible poisoning strategy is to wait for an admin action executed via `A` that violates a **C1** invariant and, thus, generates an alert. If that alert is dismissed as an intentional admin action, HOUSTON would integrate the event into its notion of "normal" behavior. As a result, future transactions exhibiting the same behavior will not be flagged, allowing the attacker to call `B` to close the market without triggering an alert. This example shows that even when poisoning is feasible, HOUSTON still raises the difficulty of successful attacks: Adversaries must align their actions with rare administrative events *and* rely on the security team inadvertently dismissing the initial warning.

VIII. Related Work

Smart Contract Security. The most effective way to protect DeFi protocols is to identify vulnerabilities during development, *before* deployment. To support this goal, the research community has explored several approaches: static analysis, formal verification, symbolic execution, and fuzzing. Static analysis is a technique used to scrutinize a program's content and structure without executing it [32], [84], [86], [31], [5], [26]. Formal verification uses mathematical methods to prove adherence to certain security properties [66], [29], [73]. Symbolic execution explores a program's behavior by executing code using symbolic inputs [4], [47], [35], [51], [82], [69], [83], [17], [56], [64]. Lastly, fuzzing involves repeatedly executing a target contract with many different generated inputs. [81], [63], [34], [36]. All these techniques have been used alone or in combination to identify both simple and complex bugs, such as integer overflows, reentrancy, excessive gas usage, and unsanitized calls. While identifying attacks before deployment is an important goal, it is often unattainable. Therefore, complementary approaches are required, such as detection (which is the focus of this work) and prevention.

Attack Detection. Several prior works detect attacks by characterizing the patterns of actions performed by a user or a smart contract [54], [92], [38], [74]. These papers primarily focus on detecting Ponzi schemes, phishing, and other scams directed toward end users. DeFiRanger is a specialized tool to detect price manipulation attacks [88]. A handful of recent works attempt to tackle the topic of generalized anomaly detection for smart contracts. In this work, we discussed `BlockGPT` and `APE` [30], [58]. Other frameworks perform on-chain anomaly detection, but the alert conditions (i.e., assertions) must be manually annotated [10], [68], [18], [91]. Conversely, HOUSTON, does not need any transaction annotations to deliver off-chain anomaly detection. Several commercial anomaly detectors have also been developed in the past few years [41], [57], [37], [28], [39]. These products generally require developers to manually define custom conditions that specify which behaviors should be permitted (or prohibited) during executions, or they rely on flagging transactions originating from suspicious sources such as sanctioned accounts or crypto mixers [80]. These conditions are evaluated either on-chain or off-chain, and any detected violation typically results in a warning or in the transaction being blocked. HOUSTON fundamentally differs from these approaches, as it eliminates the need for manually specified detection rules. Instead, HOUSTON's models automatically infer the relevant signals for a given protocol. Furthermore, HOUSTON does not depend on external metadata such as transaction provenance, which is generally an unreliable signal prone to both false positives and false negatives.

Attack Prevention. Various methods exist to prevent or mitigate an attack once it is recognized. Zhou et al. observe that about half of the DeFi protocols they surveyed include an emergency pause mechanism [96]. Other works develop automated syntheses of attack-stealing transactions, which are capable of performing the attack as "white hat" instead of the true attacker [58], [94], [89]. Notably, HOUSTON can act as the detection component that triggers such front-running tools.

Invariant Inference. `Daikon` is a system that infers likely invariants from program traces [21]. Liu et al. applied `Daikon` to smart contracts through their tool, `InvCon`, which extracts simple invariants from `ERC20` token contracts. However, `InvCon` has limitations, as it requires manual interpretation of invariant violations by a human operator. In contrast, HOUSTON provides a generic analysis to extract likely invariants from any smart contract, which can be continuously refined at runtime. This allows the invariants to adapt dynamically during live monitoring. Furthermore, HOUSTON automatically checks for violations, facilitating the timely detection of anomalous behavior. Chen et al. [11] utilized invariant templates to identify exploits related to known smart contract vulnerabilities, which requires manual annotation of template-relevant variables. In contrast, HOUSTON's models are fully automated and can detect attack transactions as anomalies without requiring adaptation to specific attack types.

IX. CONCLUSIONS

In this paper, we introduce HOUSTON, a novel anomaly detection system for identifying attacks against Ethereum-based DeFi protocols. HOUSTON focuses on detecting deviations in protocol execution rather than recognizing known vulnerability patterns. HOUSTON distinguishes itself by (1) detecting anomalies through the analysis of the protocol interactions and likely invariants, and (2) adapting and evolving by ingesting new data from transactions. For evaluation, we used a comprehensive dataset of 115 attacks on DeFi protocols. The efficacy of HOUSTON, demonstrated through a comparative analysis with state-of-the-art systems, shows its superior detection rate of 94.8% with a low false-positives rate of 0.16%. Experiments on live traffic confirm HOUSTON's efficiency and its low false-positive rate in real-world deployments.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grant numbers CNS-2334709 and IIS-2229876 and is supported in part by funds provided by the National Science Foundation, by the Department of Homeland Security, and by IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or its federal agency and industry partners.

REFERENCES

[1] Flashbots. https://www.flashbots.net/.

[2] Metamask: The ultimate crypto wallet for defi, web3 apps, and nfts. https://metamask.io/.

[3] blocksecteam. Revest finance vulnerability. https://blocksecteam.medium.com/revest-finance-vulnerabilities-more-than-re-entrancy-1609957b742f, 2022.

[4] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 161–178. IEEE, 2022.

[5] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st Conference on Programming Language Design and Implementation*, 2020.

[6] BSC. Bnb chain. https://www.bnbchain.org/en/bnb-smart-chain, 2025.

[7] CertiK. Bacon protocol incident anaylsis. https://www.certik.com/resources/blog/bacon-protocol-incident-anaylsis, 2022.

[8] chainsec. Timeline of defi exploits. https://chainsec.io/defi-hacks/.

[9] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), 2009.

[10] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, Yuxing Tang, Xiaodong Lin, and Xiaosong Zhang. SODA: A generic online detection framework for smart contracts. In *27th Annual Network and Distributed System Security Symposium, NDSS*, 2020.

[11] Zhiyang Chen, Ye Liu, Sidi Mohamed Beillahi, Yi Li, and Fan Long. Demystifying invariant effectiveness for securing smart contracts. In *ESEC/FSE 2024*, 2024.

[12] CNBC. Defi is the wild-west of crypto, 2021.

[13] Coinbase. Euler compromise investigation. https://www.coinbase.com/blog/euler-compromise-investigation-part-1-the-exploit, 2023.

[14] Coindesk. Axie infinity ronin network suffers 625m exploit, 2022.

[15] Coindesk. Nomad bridge drained of nearly 200-million in explpoit, 2022.

[16] Cointelegraph. Euler finance attack how it happened and what can be learned, 2023.

[17] ConsenSys. Mythril. https://github.com/ConsenSys/mythril, 2022.

[18] Thomas Cook, Alex Latham, and Jae Hyung Lee. Dappguard: Active monitoring and defense for solidity smart contracts. 2017.

[19] DeFiLlama. Dashboard. https://defillama.com/chain/Ethereum, 2024.

[20] dune.com. Dune. https://dune.com/, 2025.

[21] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, pages 213–224, 1999.

[22] Michael D Ernst, Adam Czeisler, William G Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd international conference on Software engineering*, pages 449–458, 2000.

[23] Ethereum. Ethereum. https://ethereum.org/en/, 2022.

[24] Ethereum. Erc-20 token standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/, 2023.

[25] evmdecompiler. evmdecompiler. https://evmdecompiler.com/, 2025.

[26] 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)*, pages 8–15, 2019.

[27] Forbes. Terrorists, north korea and other illicit actors move beyond bitcoin, 2023.

[28] Forta. What is forta firewall: Redefining onchain security, 2025.

[29] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2757–2774, 2020.

[30] Yu Gai, Liyi Zhou, Kaihua Qin, Dawn Song, and Arthur Gervais. Blockchain large language models. *arXiv preprint arXiv:2304.12749*, 2023.

[31] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. Etainter: Detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 728–739. ACM, 2022.

[32] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Analyzing the out-of-gas world of smart contracts. *Commun. ACM*, 63(10):87–95, 2020.

[33] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. Elipmoc: advanced decompilation of ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 6:1–27, 2022.

[34] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: Effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 557–560. ACM, 2020.

[35] Fabio Gritti, Nicola Ruaro, Robert McLaughlin, Priyanka Bose, Dipanjan Das, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. Confusum contractum: confused deputy vulnerabilities in ethereum smart contracts. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1793–1810, 2023.

[36] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 531–548. ACM, 2019.

[37] hexagate. Linea x hexagate: Building the world's most secure l2. https://www.hexagate.com/blog/linea-hexagate-building-the-worlds-most-secure-l2, 2025.

[38] Teng Hu, Xiaolei Liu, Ting Chen, Xiaosong Zhang, Xiaoming Huang, Weina Niu, Jiazhong Lu, Kun Zhou, and Yuan Liu. Transaction-based classification and detection approach for ethereum smart contract. *Information Processing & Management*, 58(2):102462, 2021.

[39] hypernative. Hypernative to bolster on-chain security across the linea network. https://www.hypernative.io/blog/hypernative-to-bolster-on-chain-security-across-the-linea-network, 2025.

[40] Immunefi. Hack analysis: Beanstalk governance attack. https://medium.com/immunefi/hack-analysis-beanstalk-governance-attack-april-2022-f42788fc821e, 2022.

[41] ironblocks. ironblocks. https://ironblocks.com/, 2025.

[42] Wall Street Journal. How north korea's hacker army stole $3 billion in crypto, funding nuclear program, 2023.

[43] Christopher Krügel, Thomas Toth, and Engin Kirda. Service specific anomaly detection for network intrusion detection. In *Proceedings of the 2002 ACM symposium on Applied Computing*, pages 201–208, 2002.

[44] Johannes Krupp and Christian Rossow. teEther: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, 2018.

[45] ledgerwatch. Erigon. https://github.com/ledgerwatch/erigon, 2023.

[46] Junrui Liu, Yanju Chen, Bryan Tan, Isil Dillig, and Yu Feng. Learning contract invariants using reinforcement learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–11, 2022.

[47] 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, page 254–269. ACM, 2016.

[48] Xingyu Lyu, Mengya Zhang, Xiaokuan Zhang, Jianyu Niu, Yinqian Zhang, and Zhiqiang Lin. An empirical study on ethereum private transactions and the security implications. *arXiv preprint arXiv:2208.02858*, 2022.

[49] MakerDAO. The maker protocol: Makerdao's multi-collateral dai (mcd) system. https://makerdao.com/en/whitepaper/, 2023.

[50] Neptune Mutual. Analysis of the barley finance exploit. https://medium.com/neptune-mutual/analysis-of-the-barley-finance-exploit-d2df61b98c80, 2024.

[51] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC 18, page 653–663. ACM, 2018.

[52] node real. bsc-erigon. https://github.com/node-real/bsc-erigon, 2025.

[53] OpenAI. Gpt-4.1. OpenAI API, April 2025. Large language model; context window 1 million tokens; API access.

[54] Bofeng Pan, Natalia Stakhanova, and Zhongwen Zhu. Ethershield: Time interval analysis for detection of malicious behavior on ethereum. *ACM Trans. Internet Technol.*, 2023.

[55] paradigmxyz. Blazing-fast implementation of the ethereum protocol. https://github.com/paradigmxyz/reth, 2025.

[56] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy*. IEEE, 2020.

[57] phylax. Verifiable, embedded hack prevention. https://phylax.systems/, 2025.

[58] Kaihua Qin, Stefanos Chaliasos, Liyi Zhou, Benjamin Livshits, Dawn Song, and Arthur Gervais. The blockchain imitation game. *arXiv preprint arXiv:2303.17877*, 2023.

[59] Kaihua Qin, Zhe Ye, Zhun Wang, Weilin Li, Liyi Zhou, Chao Zhang, Dawn Song, and Arthur Gervais. Towards automated security analysis of smart contracts based on execution property graph. *arXiv preprint arXiv:2305.14046*, 2023.

[60] Kaihua Qin, Liyi Zhou, Yaroslav Afonin, Ludovico Lazzaretti, and Arthur Gervais. Cefi vs. defi–comparing centralized to decentralized finance. *arXiv preprint arXiv:2106.08157*, 2021.

[61] QuillAudits. Decoding euler finance's $197 million exploit — quillaudits, 2023.

[62] Rekt. Rekt, leaderboard. https://rekt.news/leaderboard/, 2022.

[63] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. Efcf: High performance smart contract fuzzing for exploit generation. In *2023 IEEE 8th European Symposium on Security and Privacy*, pages 449–471, 2023.

[64] Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. Not your type! detecting storage collision vulnerabilities in ethereum smart contracts. In *Network and Distributed Systems Security Symposium 2024*, 2024.

[65] Rust-lang. Rust language. https://www.rust-lang.org/, 2025.

[66] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. Ethor: Practical and provably sound static analysis of ethereum smart contracts. In *CCS 20*, page 621–640, 2020.

[67] Kudelski Security. The poly network attack, 2021.

[68] R. K. Shyamasundar. A framework of runtime monitoring for correct execution of smart contracts. In *Blockchain – ICBC 2022*, pages 92–116. Springer Nature Switzerland, 2022.

[69] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language Model-Guided symbolic execution. In *30th USENIX Security Symposium*, pages 1361–1378. USENIX Association, 2021.

[70] soliditylang. Contract abi specification. https://docs.soliditylang.org/en/develop/abi-spec.html, 2023.

[71] soliditylang. Storage layout. https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_storage.html, 2023.

[72] Soliduslabs. Rug pull crypto scams. https://www.soliduslabs.com/post/rug-pull-crypto-scams, 2022.

[73] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. Smartpulse: Automated checking of temporal properties in smart contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 555–571, 2021.

[74] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing, and Baoxu Liu. Evil under the sun: understanding and discovering attacks on ethereum decentralized applications. In *30th USENIX Security Symposium*, pages 1307–1324, 2021.

[75] Tianle Sun, Ningyu He, Jiang Xiao, Yinliang Yue, Xiapu Luo, and Haoyu Wang. All your tokens are belong to us: Demystifying address verification vulnerabilities in solidity smart contracts. *arXiv preprint arXiv:2405.20561*, 2024.

[76] SunWeb3Sec. Defihacklabs. https://github.com/SunWeb3Sec/DeFiHackLabs, 2025.

[77] Adrian Tang, Simha Sethumadhavan, and Salvatore J Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *RAID 2014*, pages 109–129. Springer, 2014.

[78] Solidity Team. Solidity. https://soliditylang.org, 2022.

[79] thestreet.com. Crypto stolen in 2024 tops 1.2 billion. https://www.thestreet.com/crypto/markets/crypto-stolen-in-august-declines-but-2024-losses-already-top-1-2-billion, 2024.

[80] TornadoCash. Tornadocash. https://tornado.cash/, 2025.

[81] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 103–119, 2021.

[82] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *ACSAC 18*, page 664–676. ACM, 2018.

[83] Christof Ferreira Torres, Mathis Steichen, and Radu State. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium*, pages 1591–1607, 2019.

[84] 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*, page 67–82, 2018.

[85] Uniswap. Uniswap info. https://v2.info.uniswap.org/home, 2023.

[86] Shuai Wang, Chengyu Zhang, and Zhendong Su. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019.

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

[88] Siwei Wu, Dabao Wang, Jianting He, Yajin Zhou, Lei Wu, Xingliang Yuan, Qinming He, and Kui Ren. Defiranger: Detecting price manipulation attacks on defi applications. *arXiv preprint arXiv:2104.15068*, 2021.

[89] Yue Xue, Jialu Fu, Shen Su, Zakirul Alam Bhuiyan, Jing Qiu, Hui Lu, Ning Hu, and Zhihong Tian. Preventing price manipulation attack by front-running. In *International Conference on Artificial Intelligence and Security*, pages 309–322. Springer, 2022.

[90] ycharts. Ethereum market cap. https://ycharts.com/indicators/ethereum_market_cap, 2024.

[91] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. TXSPECTOR: Uncovering attacks in ethereum from transactions. In *29th USENIX Security Symposium*, pages 2775–2792, 2020.

[92] Yanmei Zhang, Wenqiang Yu, Ziyu Li, Salman Raza, and Huaihu Cao. Detecting Ethereum Ponzi Schemes Based on Improved LightGBM Algorithm. *IEEE Transactions on Computational Social Systems*, 9(2):624–637, 2022.

[93] 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.

[94] 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. USENIX Association, 2023.

[95] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. In *ICSE*, 2023.

[96] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2444–2461. IEEE, 2023.

Appendix A
Storage Variables Decoder

The storage layout report emitted by the compiler provides a high-level overview of the storage structure. However, this report does not allow direct mapping of a given storage access (i.e., `SSTORE`) to the associated source code variables. For example, the storage layout may show that at slot id `0` there is a static array `my_struct[100]` with 100 *contiguous* elements – where `my_struct` is a simple data structure with few fields packed in a single storage slot. Given an `SSTORE` to slot `0x3`, to understand which variables are being overwritten, one would need to automatically recognize that slot id `0x3` still falls within the definition of the static array. To perform the translation from the compiler-generated storage report to a programmatically usable data structure, we implemented an algorithm that recursively resolves every type definition within the storage layout and provides a "flat" view of the contract's storage structure.

Complex Variable Types. Complex variable types, such as mappings, and nested data structures are characterized by storage access patterns that use the `KECCAK256` hash function to derive the accessed slot id. These ids are calculated using either the accessed index (for a dynamic array) or the accessed key (for a dynamic mapping). As a result, since there are countless ways to access the same data structure, it is impractical to enumerate all of them in the storage layout report. Instead, we characterize accesses to dynamic slots by

TABLE I: Example of the `KECCAK256` look-up table collected during the transaction tracing (hashes are truncated for readability). To resolve an `SSTORE` that writes at slot id `0xd44ee5c9`, one can do a reverse-lookup in the table and extract the `BASE` in the pre-image until we identify a static slot. For example, here the look-up sequence would be: `0xd44ee5c9`→`0xbc36789e`→`0x0`. This shows the access pattern [**`0x0, KECCAK256, KECCAK256`**] that reveals access to an element of a nested mapping.

Preimage	Image
`admin.0x0`	`0xbc36789e`
`savings.0xbc36789e`	`0xd44ee5c9`

analyzing the sequence of operations that the EVM executes (during a transaction) to calculate the slot id. For example, consider the `KECCAK256` look-up table structure shown in Table I, resulting from the look-up **users**[“admin”][“savings”]. In this example, `users` is a nested data structure defined as **`map1`**`[string:`**`map2`**`[string:uint256]]`. To perform the look-up, the following operations are executed: **`KECCAK256`**`(KEY2.(`**`KECCAK256`**`(KEY1.BASE1))`, where `KEY1`= “admin”, `BASE1=0x0`, `KEY2`=“savings”, and `BASE2=0xbc36789e`. In this example, the elements nested within **map2** are uniquely identified by the access pattern [**`0x0, KECCAK256, KECCAK256`**].

Given a slot id, the (flattened) contract’s storage layout, and the `KECCAK256` look-up table collected during the transaction tracing, the function $\mathcal{SVD}$ will: (1) if the slot id corresponds to a static slot, simply return the corresponding slot information from the storage layout; otherwise, (2) if the slot id is calculated using the `KECCAK256` hash function (e.g., `0xd44ee5c9` in TableI), iteratively perform reverse-lookups of the image within the `KECCAK256` look-up table (Table I) to identify the corresponding dynamic data structure and its base slot id. Once this is identified, the $\mathcal{SVD}$ procedure returns precise slot packing information using the storage layout.

Appendix B
Dataset Overview

We observe a significant variation in the number of smart contracts in the protocols, the number of transactions, and the protocol’s lifetime at the moment of the attack. We compute the lifetime of a protocol from the first time that any of its contracts was deployed.

TABLE II: Overview of DeFi Attacks Dataset.

	Transaction # per Protocol	Contract # per Protocol	Lifespan of Protocol (days)	Date of Exploitation
mean	74,662	131	344	2022-11-24
min	5	1	1	2020-01-10
25%	776	12	54	2022-01-14
50%	7,760	42	192	2023-03-13
75%	57,282	126	534	2023-11-27
max	1,079,908	1,378	2,133	2024-09-03

Appendix C
Ablation Study

In this paper, we argue that high-quality signals are critical for obtaining accurate detection results. To support this claim, this section presents the results of two ablation studies.

Protocol Interaction versus All Interactions. First, we show how focusing on critical functions (§ V-C) – rather than on *all* function calls – in a trace is crucial to lower false positives. To this end, we rerun our experiments with an *Interaction Model* that computes fingerprints based on all function calls. Figure 7 shows the results and demonstrates that the false positives drastically increases when using the full call trace. Specifically, without focusing on critical functions, *Interaction Model* produces 192 times more false positives (1,258,841) than the paper setup that facilities the interaction simplification techniques. This highlights the importance of selective attention to relevant interactions for achieving accurate detection.

Fine- versus Coarse-Grained Storage Analysis. The ability to identify the type and layout of storage variables (fine-grained storage analysis) is important for precise invariant generation, especially when protocols pack multiple variables in the same storage slot. For example, the value `0x40301020` might represent two different `uint16` variables – `0x4030` and `0x1020` – used in two different parts of the program. Interpreting the whole slot as a single `uint256` variable (coarse-grained storage analysis) would lead to imprecise (or incomplete) invariants that negatively affect the decisions of the *Invariant Model*. To assess the impact of the analysis granularity, we modify the function $\mathcal{SVD}$ (§ V-A) to always report *one* single variable per slot of type `uint256`.

The *Invariant Model* detected four additional attacks with fine-grained storage analysis. We use the `sealfinance` incident to illustrate how coarse-grained analysis can lead to imprecise detection. The attack transaction violated the invariant in function `swap`: the argument `amountOut` must be less or equal to storage variable `reserve`. The coarse-grained analysis failed to identify this invariant because `reserve` was packed in the same slot with two other variables.

For false positive, the two configurations had similar performances, with fine-grained analysis yielding a slightly lower (0.7%) total number of false alarms. With more refined storage understanding, the precision of invariants improves. Meanwhile, the sheer number of variables and invariants also grows, which inevitably leads to an increase in false positives. These two opposing forces play out differently across protocols, and tend to balance out when considered across the entire dataset.

Appendix D
Invariant Model Hyperparameter

The *Invariant Model* leverages two hyperparameters (§V-E) to tune the confidence of invariants. For an invariant to be

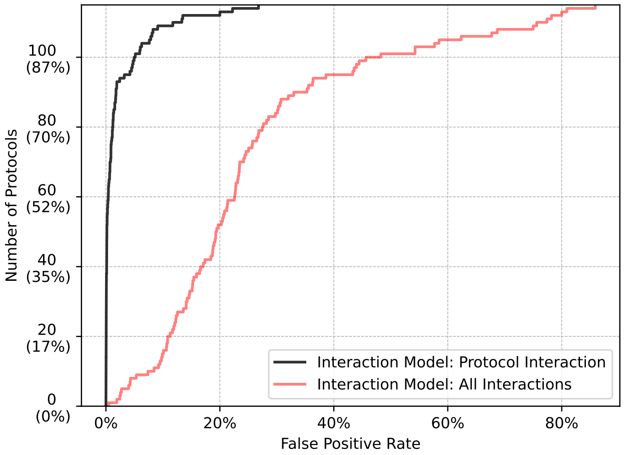

Fig. 7: Cumulative distribution of false positive rates of the *Interaction Model* with two different configurations.

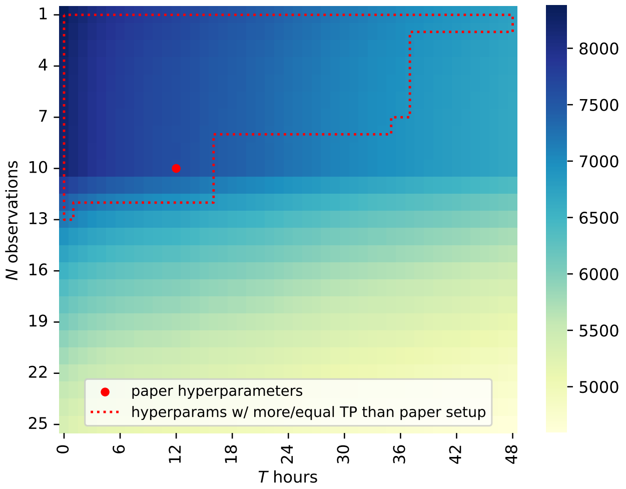

Fig. 8: Total number of false positives of the *Invariant Model* across all protocols with different hyperparameter setups. The paper setup does not overfit to the dataset.

considered valid, it has to be backed up by a certain number of transactions ($N = 10$), and the contract has to be deployed for a minimum time span ($T = 12$ hours). We chose these values with common sense and the experience with a pilot study on `euler` and `inversefinance`.

After evaluating HOUSTON on the whole dataset, we investigated our choice of hyperparameters. Interestingly, we observed that there is a wide range of reasonable setups with little effect on the attack detection result (see Figure 8). Under our setup, the *Invariant Model* happened to miss two attacks (`btfinance`, `olympusdao`) as they both had violations against invariants inferred with 9 previous transactions while we required 10 or more observations for an invariant to be considered valid. Our *Interaction Model*, and thus HOUSTON, however, caught both attacks.

Higher N and T reduce the number of false positives, while the risk of missing attacks increases. An "overfit-to-dataset" hyperparameters setup ($N = 1$, $T = 48$) will have the same detection outcome as our more generic setup while yielding 11% fewer false alarms. Of course, we kept our initial setup and reported all results in the generic setting.

APPENDIX E
LIVE EXPERIMENTS PERFORMANCE STATISTICS

HOUSTON performs the following procedures during operation: (1) `Trace`: Upon spotting a new transaction, HOUSTON collects the execution trace (using our custom plugin) from the Ethereum node. (2) `Process`: HOUSTON parses the raw execution trace, extracts relevant information, and turns it into the format needed for further analysis. (3) `Detect`: The *Interaction Model* and the *Invariant Model* check for anomaly and raise warnings if necessary. HOUSTON then integrates new transaction data into its models.

The time statistics for these procedures, along with the end-to-end execution time (the cumulative time taken by steps (1)-(3)), are detailed in Table III. We also provide statistics about the load of Ethereum mainnet traffic HOUSTON handled during the live experiments in Table IV.

TABLE III: HOUSTON execution time (in second).

	Trace	Process	Detect	End-to-End
mean	0.003	0.180	0.028	0.211
median	0.003	0.133	0.013	0.148
99%	0.006	1.105	0.191	1.165
max	0.337	2.912	0.810	3.085

TABLE IV: Transaction load for HOUSTON during live experiments (number of transactions per second).

	Filter	Trace	Process / Detect
mean	24.9	12.5	0.1
median	24.2	12.1	0.0
99%	46.4	24.1	0.4
max	725.0	362.9	2.2

APPENDIX F
HIGH TVL PROTOCOLS

The boundaries of the protocols listed in Table V are approximated starting from the reference contract as discussed in §VI-A. We picked as *reference contract* the one listed on *defillama.com* [19].

TABLE V: Evaluation results on high TVL protocols with *no known* history of exploitation. `Duration` is measured in days.

	Protocol	**Tot TXs**	**Duration**	**FP**	(%)
1	ethena	589,317	123	12	(0.0%)
2	renzo	230,479	94	17	(0.0%)
3	etherfi	494,229	136	19	(0.0%)
4	ondofinance	496,387	821	24	(0.0%)
5	stakestone	417,500	342	63	(0.0%)
6	aave	2,558,803	1,406	120	(0.0%)
7	uniswap	2,868,003	1,412	132	(0.0%)
8	rocketpool	944,573	1,030	288	(0.0%)
9	stargatefinance	1,754,758	863	301	(0.0%)
10	everclear	41,491	480	42	(0.1%)
11	uncxnetwork	246,268	1,381	133	(0.1%)
12	stakewise	99,950	1,278	140	(0.1%)
13	fraxfinance	179,405	665	244	(0.1%)
14	pendle	458,250	1,186	384	(0.1%)
15	tethergold	46,787	1,235	75	(0.2%)
16	aura	207,711	779	859	(0.4%)
17	convexfinance	1,845,533	1,166	7,396	(0.4%)
18	beefy	11,970	338	66	(0.6%)
19	hasnote	5,007	423	175	(3.5%)
	Total	13,496,421	-	10,488	(0.08%)

APPENDIX G
ADDITIONAL TRUE POSITIVE EXAMPLES

Polynetwork. The attack was detected by a violation of a class **C2** invariant of the argument `curEpochPkBytes` in the function `putCurEpochConPubKeyBytes` callsite. This is in direct correlation with the attack vector: The attacker exploited a confused deputy vulnerability [35] to call putCurEpochConPubKeyBytes and modify the admin keys [67].

Audius. The attack violated three **C2** and two **C3** invariants tied to governance settings, which were aggressively altered to enable malicious proposals. For example, the attacker set undelegateLockupDuration to 1, breaking its historical parity with removeDelegatorLockupDuration.

Coverprotocol. The attack was detected by the *Invariant Model* as a violation of a class **C2** invariant involving the variables _totalSupply and _amount when entering the function mint(). Historically, during minting, _amount was always less than _totalSupply. However, in this attack, over 40 quintillion COVER tokens were minted, causing the invariant to be violated.

Baconprotocol. The attack was detected by the *Interaction Model*. Specifically, the sequence of actions lend-lend-lend-redeem was never observed before in the history of the protocol’s execution. This strongly correlates with the root cause issue of the attack: a reentrancy bug in the lend() function [7].

BarleyFinance. The attack was detected by both our models. The *Interaction Model* registered a very unique and long sequence of repeated actions (flash-bond)^n which exactly match the attack technique: a *flash-loan-based reentrancy attack* [50]. On the other hand, the *Invariant Model* spotted a violation of a **C3** invariant: the value of _totalSupply was suddenly lower than the _amount arguments in the final call to the function debond.

Appendix H
LLM prompts

The following are the system and user prompts we used to decide if the comparison between two variables in a smart contract is semantically meaningful or not.

Prompt 1: System Prompt

````
You are an expert smart contract developer
and auditor.
Your language of expertise is
Solidity/Vyper.
You will be given the code of a smart
contract, followed by two variables that
appear in the code.
For instance, you might be given:
VAR1: totalSupply
VAR2: balance
Your task is to decide if it makes sense to
compare the two variables in the context of
the smart contract.
For instance, if they are referring to two
completely different concepts, then the
relationship is not meaningful.
*** IMPORTANT ***
1. YOU MUST EMIT A REPORT IN THE FORMAT
SPECIFIED BELOW.
2. THE SUMMARY SHOULD EXPLAIN THE REASONING
BEHIND THE DECISION.
3. I WANT YOU TO BE CONSERVATIVE IN YOUR
ASSESSMENT, IN OTHER WORDS, DISCARD
RELATIONSHIPS THAT ARE NOT CLEARLY
MEANINGFUL. IF YOU ARE NOT SURE, ANSWER
YES.
Let’s think step by step.
````

Prompt 2: User Prompt

~~~
You are given this smart contract code:
<contract>
TEMPLATED_CONTRACT_CODE
</contract>
Are these two variables semantically
related and potentially comparable in the
context of the smart contract?
<variables>
TEMPLATED_RELATION
</variables>
After completing the analysis, you MUST
output a report in the specified format.
```
<report>
<verdict>
</verdict>
<summary>
...
</summary>
</report>
```
~~~

APPENDIX I
HOUSTON RESULTS OVERVIEW

TABLE VI: HOUSTON Historical Evaluation Results. ✓ represents true positive, ✗ represents false negative. Column `Tot TXs` is the total amount of transactions available for the protocol. Column `Lifespan` represents the number of days the protocol had been active prior to the attack. Column `FP` shows the total amount (and percentage) of false positive transactions in the `Tot TXs`. Column `FP/d` shows the average daily false positive alerts for a protocol. Column `Inv. FP` is the absolute number of false positives reported by the *Invariant Model*. Column `Int. FP` reports the absolute number of false positives reported by the *Interaction Model*. Column `Inv.` and `Int.` represents the decision of each model regarding the attack transaction. Column HOUSTON is the final decision taken by our system. The `BlockGPT` [30], `APE` [58], `TxSpector` [91], and `DeFiRanger` [88] columns show the results of corresponding systems. Symbol - means that the protocol was not in their dataset (when we do not have access to the tool to apply to our dataset). Specifically, for `BlockGPT`, ✓$_1$ shows that the attack was ranked as the most abnormal transaction, ✓$_2$ as the second most abnormal transaction, and ✓$_3$ as the third most abnormal one. Column `Longest` reports the results of the transaction length baseline. Finally, we use the symbol ⋆ to identify protocols that were part of our live monitoring evaluation.

	Protocol	Tot TXs	Lifespan	FP	(%)	FP/d	Inv. FP	Int. FP	Inv.	Int.	HOUSTON	BlockGPT	APE	TxSpector	DeFiRanger	Longest
1	polynetwork ⋆	57,140	363	3	(0.0%)	0.0	0	3	✓	✗	✓	✗	✗	✗	✗	✗
2	grok	66,790	6	7	(0.0%)	1.2	0	7	✗	✓	✓	-	-	✓	✗	✗
3	teamfinance ⋆	30,619	456	15	(0.0%)	0.0	0	15	✗	✓	✓	-	-	✗	✗	✓
4	mcc	59,801	536	26	(0.0%)	0.0	12	14	✓	✓	✓	-	-	✓	✓	✓
5	loopring ⋆	248,781	1142	36	(0.0%)	0.0	13	26	✓	✗	✓	-	-	✗	✓	✗
6	umbrellanetwork ⋆	139,042	405	42	(0.0%)	0.1	27	15	✓	✗	✓	-	✗	✗	✗	✗
7	shanshu ⋆	173,121	91	42	(0.0%)	0.5	28	15	✗	✓	✓	-	-	✗	✓	✗
8	audius	268,109	640	77	(0.0%)	0.1	45	33	✓	✓	✓	-	✗	✗	✗	✗
9	kashi ⋆	753,725	531	128	(0.0%)	0.2	35	97	✓	✓	✓	-	-	✗	✓	✗
10	sandboxland	1,079,908	833	231	(0.0%)	0.3	105	134	✗	✓	✓	-	✗	✗	✗	✗
11	game	4,276	1	6	(0.1%)	6.0	0	6	✗	✓	✓	-	-	✓	✗	✓
12	dexible	22,609	1074	24	(0.1%)	0.0	5	19	✗	✓	✓	-	-	✗	✗	✗
13	btc20	55,054	66	30	(0.1%)	0.5	12	21	✗	✓	✓	-	-	✗	✓	✓
14	compounderfinance	40,993	942	37	(0.1%)	0.0	20	20	✓	✓	✓	-	-	✗	✓	✗
15	kyberswap	59,986	767	42	(0.1%)	0.1	20	22	✗	✓	✓	-	-	✗	✗	✗
16	nfttrader	42,115	998	45	(0.1%)	0.0	23	22	✓	✓	✓	-	-	✗	✗	✗
17	balancer	61,931	31	57	(0.1%)	1.8	16	41	✓	✓	✓	✗	-	✓	✓	✓
18	coverprotocol	105,424	64	61	(0.1%)	1.0	18	45	✓	✗	✓	✗	-	✗	✗	✗
19	sealfinance	65,593	53	96	(0.1%)	1.8	73	25	✓	✗	✓	-	-	✓	✓	✗
20	thorchain ⋆	213,250	191	120	(0.1%)	0.6	93	30	✓	✗	✓	✗	-	✗	✗	✗
21	harvestfinance ⋆	217,992	56	151	(0.1%)	2.7	137	16	✗	✓	✓	✗	-	✓	✓	✓
22	indexedfinance ⋆	211,104	317	214	(0.1%)	0.7	89	132	✓	✓	✓	✓$_3$	✗	✓	✗	✓
23	mimspell	390,419	980	376	(0.1%)	0.4	115	280	✓	✓	✓	-	-	✗	✗	✗
24	wiselending	382,780	2133	415	(0.1%)	0.2	173	259	✗	✓	✓	-	-	✗	✗	✗
25	valuedefi ⋆	655,019	90	489	(0.1%)	5.4	384	110	✓	✓	✓	✓$_1$	-	✗	✓	✗
26	bzxprotocol	365,787	1239	515	(0.1%)	0.4	338	199	✗	✓	✓	-	-	✗	✓	✗
27	visorfinance	1,745	110	3	(0.2%)	0.0	0	3	✗	✓	✓	✓$_1$	✗	✗	✗	✓
28	tinu	3,668	586	6	(0.2%)	0.0	1	5	✓	✓	✓	-	-	✗	✓	✗
29	shoco	7,760	594	16	(0.2%)	0.0	12	6	✓	✓	✓	-	-	✓	✓	✗
30	ktaf	37,104	1291	89	(0.2%)	0.1	40	50	✓	✓	✓	-	-	✗	✓	✓
31	conicfinance ⋆	47,208	477	111	(0.2%)	0.2	53	67	✓	✓	✓	-	-	✗	✗	✓
32	klondikefinance	60,204	231	129	(0.2%)	0.6	98	38	✓	✓	✓	✓$_1$	✓	✗	✗	✗
33	alphafinance ⋆	57,424	130	135	(0.2%)	1.0	98	48	✓	✓	✓	✗	-	✗	✗	✗
34	floorprotocol	106,524	65	226	(0.2%)	3.5	191	35	✗	✓	✓	-	-	✗	✗	✗
35	inversefinance ⋆	139,687	478	247	(0.2%)	0.5	136	123	✗	✓	✓	✓$_2$	✗	✗	✓	✗
36	creamfinance ⋆	587,106	393	958	(0.2%)	2.4	717	275	✗	✓	✓	✗	✗	✗	✗	✗
37	penpiexyzio	844,841	688	2,010	(0.2%)	2.9	1,333	713	✗	✓	✓	-	-	✗	✓	✗
38	deeznutz	860	1	3	(0.3%)	3.0	0	3	✗	✓	✓	-	-	✗	✗	✓
39	btfinance	11,483	69	38	(0.3%)	0.6	20	18	✗	✓	✓	✓$_1$	-	✗	✓	✓
40	xstableprotocol	15,735	563	48	(0.3%)	0.1	26	24	✓	✓	✓	-	-	✗	✗	✓
41	revestfinance ⋆	22,438	187	57	(0.3%)	0.3	30	31	✓	✓	✓	✓$_1$	✗	✓	✗	✗
42	floordao	25,605	1081	78	(0.3%)	0.1	27	56	✓	✓	✓	-	-	✓	✗	✓
43	saddlefinance	47,308	466	132	(0.3%)	0.3	95	43	✗	✓	✓	✗	✓	✓	✗	✗
44	uniclynft	56,558	900	149	(0.3%)	0.2	77	80	✗	✓	✓	-	-	✓	✗	✗
45	hypr	1,137	29	5	(0.4%)	0.2	0	5	✓	✓	✓	-	-	✗	✗	✓
46	veth	7,729	52	33	(0.4%)	0.6	21	13	✓	✗	✓	✓$_3$	-	✓	✗	✗
47	raftfi	31,214	750	128	(0.4%)	0.2	69	70	✗	✓	✓	-	-	✗	✓	✓
48	euler ⋆	101,868	469	421	(0.4%)	0.9	257	182	✓	✓	✓	-	-	✗	✗	✗
49	0x0dex	2,590	106	12	(0.5%)	0.1	8	4	✓	✓	✓	-	-	✗	✗	✓

Continued on next page

TABLE VI – continued from previous page

	Protocol	Tot TXs	Lifespan	FP	(%)	FP/d	Inv. FP	Int. FP	Inv.	Int.	Houston	BlockGPT	APE	TxSpector	DeFiRanger	Longest
50	sodafinance	7,566	2	40	(0.5%)	20.0	28	16	✗	✓	✓	✗	-	✗	✗	✗
51	dodo ⋆	19,218	46	94	(0.5%)	2.0	42	55	✓	✓	✓	✓$_1$	-	✗	✗	✗
52	picklefinance	33,367	54	182	(0.5%)	3.4	139	56	✓	✓	✓	✗	-	✗	✗	✗
53	qtntoken	1,877	541	12	(0.6%)	0.0	5	8	✗	✓	✓	-	-	✗	✗	✓
54	thenftv2	14,194	923	88	(0.6%)	0.1	50	45	✗	✓	✓	-	-	✗	✗	✗
55	prismafi	134,389	248	871	(0.6%)	3.5	144	743	✗	✓	✓	-	-	✗	✗	✗
56	bbt	140	54	1	(0.7%)	0.0	0	1	✓	✓	✓	-	-	✓	✗	✗
57	jay	1,000	747	8	(0.8%)	0.0	1	7	✓	✓	✓	-	-	✗	✗	✗
58	xtoken	9,206	116	73	(0.8%)	0.6	33	42	✓	✓	✓	✗	✗	✓	✗	✓
59	zenterest	85,332	1325	716	(0.8%)	0.5	651	82	✗	✓	✓	-	-	✗	✗	✗
60	pineprotocol	10,749	684	92	(0.9%)	0.1	25	70	✗	✓	✓	-	-	✗	✗	✗
61	n00dtoken	2,016	49	22	(1.1%)	0.4	12	10	✓	✓	✓	-	-	✓	✗	✓
62	popsiclefinance	3,909	40	43	(1.1%)	1.1	38	5	✓	✓	✓	✓$_1$	✓	✗	✗	✓
63	blueberryprotocol	1,520	32	19	(1.2%)	0.6	9	11	✓	✓	✓	-	-	✓	✗	✗
64	dfxfinance	3,243	344	39	(1.2%)	0.1	10	29	✗	✓	✓	-	-	✓	✗	✗
65	olympusdao ⋆	3,557	254	44	(1.2%)	0.2	5	40	✗	✓	✓	-	-	✗	✗	✗
66	convergence	11,960	184	147	(1.2%)	0.8	46	105	✓	✗	✓	-	-	✗	✗	✗
67	deusdao	1,339	209	18	(1.3%)	0.1	9	12	✓	✗	✓	-	-	✗	✗	✗
68	barleyfinance	2,294	3	29	(1.3%)	9.7	10	19	✓	✓	✓	-	-	✗	✗	✓
69	affinedefi	4,532	523	60	(1.3%)	0.1	11	51	✓	✓	✓	-	-	✗	✗	✗
70	paraspacenft	40,264	106	574	(1.4%)	5.4	72	518	✓	✓	✓	-	-	✗	✓	✗
71	luckytigernft	579	28	10	(1.7%)	0.4	4	6	✗	✓	✓	-	-	✗	✗	✓
72	fulcrum ⋆	20,309	421	349	(1.7%)	0.8	270	81	✗	✓	✓	✗	-	✗	✗	✗
73	filxdn404	339	107	6	(1.8%)	0.1	0	6	✓	✓	✓	-	-	✗	✗	✓
74	sturdyfinance	8,479	384	159	(1.9%)	0.4	75	99	✓	✓	✓	-	-	✗	✓	✓
75	auctus	4,617	675	98	(2.1%)	0.1	58	44	✓	✓	✓	-	✓	✗	✗	✓
76	punkprotocol	322	113	7	(2.2%)	0.1	2	5	✓	✓	✓	✓$_1$	✗	✗	✗	✓
77	gain	1,574	581	34	(2.2%)	0.1	12	28	✗	✓	✓	-	-	✓	✗	✓
78	zunamiprotocol	5,850	522	127	(2.2%)	0.2	44	91	✓	✗	✓	-	-	✗	✓	✗
79	baconprotocol	1,148	220	27	(2.4%)	0.1	5	22	✗	✓	✓	-	✗	✗	✗	✗
80	warpfinance	475	2	12	(2.5%)	6.0	4	8	✗	✓	✓	✓$_1$	-	✗	✓	✓
81	rubic	725	122	18	(2.5%)	0.1	9	9	✓	✓	✓	-	-	✗	✗	✓
82	monoxfinance	1,066	46	27	(2.5%)	0.6	14	13	✓	✓	✓	-	✗	✗	✗	✓
83	xcarnival	1,149	21	30	(2.6%)	1.4	17	15	✗	✓	✓	-	✗	✗	✗	✓
84	paraswap	686	7	19	(2.8%)	2.7	6	13	✗	✓	✓	-	-	✗	✗	✗
85	wildcredit	2,913	33	83	(2.8%)	2.5	37	50	✓	✓	✓	✓$_1$	-	✗	✗	✗
86	cheesebank	1,800	41	52	(2.9%)	1.3	37	16	✗	✓	✓	✓$_1$	-	✗	✓	✓
87	juice	93	1	3	(3.2%)	3.0	0	3	✗	✓	✓	-	-	✗	✗	✗
88	astridprotocol	371	22	15	(4.0%)	0.7	8	9	✗	✓	✓	-	-	✓	✗	✓
89	uwerx	145	161	6	(4.1%)	0.0	4	2	✓	✓	✓	-	-	✗	✗	✗
90	templedao	545	133	25	(4.6%)	0.2	5	23	✓	✓	✓	-	-	✗	✗	✗
91	particletrade	303	161	16	(5.3%)	0.1	2	15	✓	✓	✓	-	-	✗	✗	✗
92	peapodsfinance	712	641	38	(5.3%)	0.1	8	34	✗	✓	✓	-	-	✗	✗	✗
93	ruggedart	116	10	7	(6.0%)	0.7	0	7	✗	✓	✓	-	-	✓	✗	✗
94	gooddollar	778	752	48	(6.2%)	0.1	16	35	✗	✓	✓	-	-	✓	✗	✓
95	hopelend	538	73	34	(6.3%)	0.5	11	25	✓	✓	✓	-	-	✗	✗	✓
96	omninft	255	26	19	(7.5%)	0.7	3	16	✓	✓	✓	-	✗	✗	✗	✓
97	levusdc	118	213	9	(7.6%)	0.0	3	7	✗	✓	✓	-	-	✓	✗	✗
98	dappsocial	109	59	9	(8.3%)	0.2	0	9	✗	✓	✓	-	-	✗	✗	✓
99	vinu	22	1	2	(9.1%)	2.0	0	2	✗	✓	✓	-	-	✗	✓	✓
100	pawnfi	775	55	77	(9.9%)	1.4	40	40	✓	✓	✓	-	-	✗	✗	✓
101	mahalend	223	334	23	(10.3%)	0.1	8	17	✓	✓	✓	-	-	✗	✓	✓
102	earningfram	635	306	71	(11.2%)	0.2	23	50	✓	✓	✓	-	-	✗	✗	✓
103	shata	556	140	64	(11.5%)	0.5	21	45	✗	✓	✓	-	-	✗	✗	✗
104	cowswap	298	206	35	(11.7%)	0.2	0	35	✗	✓	✓	-	-	✗	✗	✗
105	baocommunity	89	134	12	(13.5%)	0.1	1	12	✗	✓	✓	-	-	✗	✓	✓
106	onyxprotocol	5	6	1	(20.0%)	0.2	0	1	✗	✓	✓	-	-	✓	✗	✓
107	pandora	9	1	2	(22.2%)	2.0	0	2	✗	✓	✓	-	-	✗	✗	✓
108	yodlrouter	45	54	10	(22.2%)	0.2	4	6	✗	✓	✓	-	-	✗	✗	✗
109	burntbubba	56	985	15	(26.8%)	0.0	0	15	✗	✓	✓	-	-	✗	✓	✗
110	minerercx	20,844	3	15	(0.1%)	5.0	3	12	✗	✗	✗	-	-	✗	✗	✓
111	nomad	63,880	203	40	(0.1%)	0.2	24	21	✗	✗	✗	-	-	✗	✗	✗
112	azukidao	3,182	1	6	(0.2%)	6.0	4	2	✗	✗	✗	-	-	✗	✗	✗
113	snood	12,042	347	42	(0.3%)	0.1	30	12	✗	✗	✗	-	✗	✗	✗	✗
114	88mph ⋆	14,257	509	198	(1.4%)	0.4	108	92	✗	✗	✗	✓$_1$	-	✗	✗	✗
115	bzx	32,553	456	440	(1.4%)	1.0	352	90	✗	✗	✗	✗	-	✓	✓	✗
	TOTAL	8,586,421		13,827	(0.16%)		7,702	6,568								