Quantitative Runtime Monitoring of Ethereum Transaction Attacks

Anonymous Author(s)

Abstract

The rapid growth of decentralized applications, while revolutionizing financial transactions, has created an attractive target for malicious attacks. Existing approaches to detecting attacks often rely on predefined rules or simplistic and overly-specialized models, which lack the flexibility to handle the wide spectrum of diverse and dynamically changing attack types.

To address this challenge, we present a general and extensible framework, MoE (**Mo**nitoring **E**thereum), that leverages *runtime verification* to detect a wide range of attacks on Ethereum. MoE features an expressive attack modeling language, based on Metric First-order Temporal Logic (MFOTL), that can formalize a wide range of attacks. We integrate a novel semantic lifting approach that extracts vital system behaviors for various attacks utilizing the monitoring tool MonPoly. Furthermore, we further equip MoE with quantitative capabilities to evaluate the similarity between a transaction and an attack formula to empower the performance in identifying attacks, including near-miss attacks.

We carry out extensive experiments with MoE on a labeled benchmark and a large-scale dataset containing over one million transactions. On the labeled benchmark, MoE successfully detects 92.0% attacks and achieves a 45.0% higher recall rate than competing state-of-the-art tool. MoE finds 3,319 attacks with 95.4% precision on the large dataset. Furthermore, MoE uses quantitative analysis to uncover 8% more attacks. Finally, the average time for monitoring a transaction is less than 23 ms, positioning MoE as a promising practical solution for real-time attack detection for Ethereum.

Keywords

Ethereum, Runtime Monitoring, Ethereum Attack Detection

1 Introduction

In the realm of blockchain and smart contract technologies, the decentralized application (DApp) ecosystem has gained substantial attention [22–24]. Smart contracts are now widely used, in particular in financial sector [10], and manage a wide range of assets [18]. Ethereum, the driving force behind these innovative applications, has witnessed a remarkable increase in its market capitalization [1]. Such milestones underscore the vast potential of this ecosystem, marking it a key area of development in the world of digital finance.

Unfortunately, this surge has also brought forth a darker reality: transaction-level attacks, which result in illegal financial gains on Ethereum, are becoming a trend [18]. The ValueDeFi incident[14] exemplifies the severe impacts of such attacks, where an attacker exploited the MultiStables library via a flash loan, causing a loss of 6 million USD [14]. Recently, innovative transaction-level attacks such as call injection and sandwich have emerged, undermining transaction integrity and manipulating market outcomes [5, 25]. Specifically, call injection attacks perturb smart contract operations by altering function calls, resulting in unauthorized transactions. Sandwich attacks, particularly prevalent in decentralized exchanges, reorder transactions for profit [25]. These attacks exploit transaction-level "bugs", thus bypassing conventional code-level vulnerability detection methods. Additionally, some code-level vulnerabilities [9, 12, 19, 20, 27], like reentrancy, also manifest at the transaction level, underscoring the need for new methodologies to bolster Ethereum's security.

Related Work. Several approaches have been proposed for detecting transaction-level attacks in Ethereum. DEFIER [17] employs a sequence-based classifier, in the form of a *multilayer perceptron*, to supplement the missing attack information and reconstruct execution traces for each incident. *Zhou et al.* [25] formalize sandwich attacks, which involve front-running and back-running transactions on an exchange. *Qin et al.* [13] propose to quantify the blockchain maximal extractable value (MEV) as a way of detecting such attacks. *Daian et al.* [6] first introduced the concept of MEV to detect potential transaction re-ordering attacks. *Wu et al.* [21] introduced DeFiRanger, which detects price manipulation attacks using patterns with lifted DeFi semantics.

These approaches are, however, far from satisfactory. First, they are not always effective. For instance, most of them fail to detect attacks that involve a single exploit transaction with limited profit. Second, the existing approaches are designed for a specific type of attack and their scope cannot easily be extended. For instance, some of the approaches [21, 25] are exclusively limited to detecting price manipulation attacks. These shortcomings underscore the importance of developing a general and systematic framework for attack finding, encompassing both the previously mentioned ones and potential future unknown threats.

Challenges. To advance this area, we tackle this problem using runtime monitoring. Runtime monitoring is a lightweight, and highly effective formal method where one can specify desired or malicious behaviors, and monitor their occurrence in real-time, at system runtime. To apply runtime verification to detecting transaction-level attacks, we must address several nontrivial technical challenges. *First*, we must develop formal models that allow us to precisely capture all the existing transaction-level attacks that we aim to monitor, since the precision of the model here is critical to minimize false alarms. And, the models should be sufficiently abstract to ensure generality while minimizing the monitoring overhead. *Second*, we need to develop a semantic lifting method that efficiently extracts relevant information from raw Ethereum transaction logs so as to enhance the practicality of the approach. Meanwhile, the semantic lifting should be flexible enough to extract further relevant information as needed to ensure our framework can support future attacks. *Lastly*, our approach must be efficient to enable real-time monitoring.

Introducing MoE. We propose MoE, a general and extensible framework for runtime **Mo**nitoring of transaction-level attacks in **E**thereum, with built-in support for many types of popular attacks. To tackle the above-mentioned challenges, *first*, we develop intuitive formal models to capture the semantics of transaction logs for Ethereum. *Second*, we propose a systematic semantic lifting approach to automatically extract the system behaviors from these logs in a way suitable for runtime monitoring. *Lastly*, we specify the behavior of different types of attack at an appropriate level of abstraction. Although the attacks differ substantially in how they exploit vulnerabilities, all the attacks covered exhibit similar temporal behavior. Using MFOTL properties to describe the attacks and the relevant information extracted through semantic lifting, our framework then deploys the state-of-the-art runtime monitoring engine, MonPoly [3], to detect transaction-level attacks on Ethereum. Furthermore, we extend MoE with the capability of quantitative analysis. This not only enables the monitoring of a broader spectrum of attacks, it also potentially opens the door for future runtime enforcement (so that we can prevent attacking from happening in time).

Evaluation. We evaluate MoE using a labeled benchmark with 24 attacks and a large-scale dataset containing more than one million Ethereum transactions. The experimental results show that MoE successfully detects 22 attacks in the benchmark, achieving an average recall of 92.0%. In the large-scale dataset, MoE successfully detects 3,319 attack transactions with a precision rate of 95.0%. In particular, these attacks cause 118.19 million USD of financial loss in total, and most of them have gone previously unnoticed. Moreover, quantitative analysis improves effectiveness by detecting 8% more attacks with an acceptable false positive rate. In addition, the average time needed to monitor a transaction is less than 23 ms, which is substantially less than the execution time of Ethereum transactions. Hence MoE is capable of real-time monitoring.

Contributions. In summary, we make the following contributions.

- We introduce MoE, a general and extensible runtime monitoring framework tailored for detecting transaction-level attacks on Ethereum. Supporting a new type of attack in our framework is as simple as specifying a temporal logic formula that models the attack.
- We propose a systematic and extensible semantic lifting approach that extracts concise information from raw system logs that characterize the behavior of the system for runtime monitoring.
- We demonstrate the capabilities of MoE by formalizing and detecting five prominent kinds of transaction-level attacks across large-scale datasets. Our experimental results highlight our framework's effectiveness and efficiency, showcasing the potential of using runtime verification to detect a wide range of attacks on Ethereum.

2 Background

In this section, we provide background on the Ethereum and transaction-level attacks. We also give a brief introduction to MonPoly, which is used for runtime verification in our framework.

2.1 Blockchain and Transaction-level Attacks

Blockchain. In a blockchain system, an *account* is a digital entity that holds and manages assets or information. The *address* of an account serves as an identity, visible to others on the network. There are two types of accounts: *External Owned Accounts (EOAs)* and *Contract Accounts (CAs)*, both of which store the ether balance. Moreover, a CA also stores the code and related storage of the smart contract, which provides multiple executable functions for handling business processes. *Transactions* serves as the entry for calling the smart contract, referring to the transfer of *asset* or data from one participant (sender) to another (receiver). There are two kinds of transactions: *external transactions* (initiated by a EOA) and *internal transactions* (initiated by a CA). In particular, since a CA can also initiate a transaction, a transaction calling a CA can derive multiple transactions. To depict a transaction, multiple components are given as:

- sender (`sdr`) and receiver (`rcv`) addresses, which identify the initiator of the transaction and the recipient who receives the assets, respectively.
- asset (`ast`), which refers to the digital representation of a value that can be transferred or exchanged within the network, taking various forms.
- the amount of the asset (`amt`), which specifies the quantity of assets being transferred.
- the invoked function (`func`) and its parameters (`params`), which serve as the instructions for smart contracts to execute specific actions or operations based on the transaction.

Transaction-Level Attacks against Ethereum. We focus on the detection of attacks against Ethereum at the transaction level in this work. For an extensive background on such attacks, we refer the reader to [26]. Transaction-level attacks refer to attacks that target the interactions between transactions and the Ethereum network's state. These attacks exploit vulnerabilities at the level of transaction execution, smart contract interactions, or the Ethereum protocol itself. These types of attacks can be subtle and difficult to detect as they often exploit the intended functionality of smart contracts rather than explicit coding errors.

Consider a simple sandwich attack. The attacker places two ordered asset exchange transactions to sandwich a normal transaction, with the former altering the blockchain state to increase the exchange rate, and the latter profiting from the state change. Another detailed example of a price manipulation attack, which is a typical type of transaction-level attack, is demonstrated in Appendix A.

2.2 Runtime Verification with MonPoly

We formulate various transaction-level attacks using Metric First-Order Temporal Logic (MFOTL) [3], an expressive logic that can capture real-time event-parameter relationships.

Full MFOTL syntax and semantics are detailed in [4]. We introduce the minimal notations used herein. Let $Sign = (C, R, l)$ be a signature. C is a set of constant symbols, R is a set of predicate symbols (for relation) and l is an arity function defined over the relation symbols. V denotes a countably infinite set of variables, and we assume $V \cap R = \emptyset$ and $V \cap C = \emptyset$. The MFOTL formulas over $Sign$ are defined as follows: (i) For $t, t' \in V \cap C$, $t = t'$ and $t < t'$ are

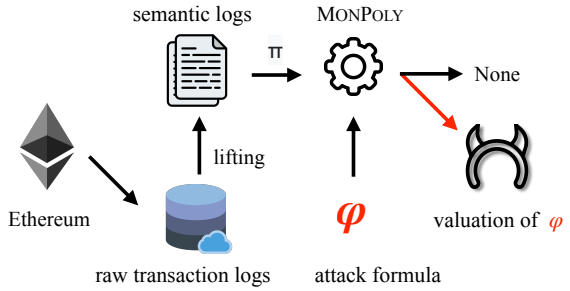


Figure 1: An Overview of MoE .

formulas; (ii) For $r \in R$ and $t_1, ..., t_{l(r)} \in V \cap C$, $r(t_1, ..., t_{l(r)})$ is a formula. (iii) For $x \in V$ if ϕ_1 and ϕ_2 are formulas then $\neg\phi_1$, $\phi_1 \wedge \phi_2$ and $\exists x.\phi_1$ are formulas; (iv) For $I \in \mathbb{I}$, if ϕ_1 and ϕ_2 are formulas then $\Diamond_I \phi_1$ (eventually), $\blacklozenge_I \phi_1$ (once), $\phi_1 \mathcal{S}_I \phi_2$ (since) and $\phi_1 \mathcal{U}_I \phi_2$ (until) are formulas. The interval set $\mathbb{I}$ is defined as $[a, b] \in \mathbb{I}$ if $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \infty$ and $a \leq b$. The operators $\Diamond_I$, $\blacklozenge_I$, $\mathcal{S}_I$ and $\mathcal{U}_I$ are augmented with an interval I, which defines the satisfaction of the formula within a time range specified by I relative to the current *timestamp* at τ_i, where $\tau_i \in \tau$ and τ is the set of timestamps.

3 The MoE Framework

Overview. MoE is designed to handle the aforementioned challenges, monitoring the transaction-level attacks effectively and compositionally. Figure 1 shows the overall workflow of MoE, which consists of three stages: *Transaction Tracing*, *Semantic Lifting*, and *Attack Detection*. In Stage 1, given the Ethereum transactions, we execute them with an instrumented EVM to obtain their system logs [16]. In Stage 2, we systematically transform the raw system logs into *semantic logs* suitable for monitoring. In Stage 3, we equip MonPoly with the semantic logs and our proposed attack formula in MFOTL for attack detection.

We further extend the capability of MFOTL and propose a quantitative semantic for a subset of MFOTL (for characterizing attacks) to evaluate the risk of attacks in a more informed way.

Notation. We begin by explaining the notation that we will use.

A *raw transaction log* refers to the input of our framework, which is a segment of undecoded JSON files.

A *system log* s consists of serialized and decoded dictionary data c from a raw transaction log. We introduce a Domain-Specific Language (Section 3.2), to precisely define the semantics of system logs using a comprehensive set of derivation rules.

A function call f in a smart contract may invoke specific functions that, within a blockchain system, can trigger a transaction event—a mechanism to record particular actions or changes. In this work, we define two disjoint sets L_t and L_e to indicate the type of statement c in a system log s: L_c for JSON data related to function calls, and L_e for data concerning transaction events.

A *semantic log* $\Pi = \langle \pi_1, ..., \pi_n \rangle$ is a stream of *events*. *Event* is denoted as $e(d_1, ...d_n)$ where e is the name and d_i represents a parameter. π_i is a set of *events*, which are considered to happen in parallel. We define two sets of predicate symbols for e: *BTS* (*basic transaction semantics*) and *AAS* (*advanced attack semantics*), which we introduc in Section 3.2.

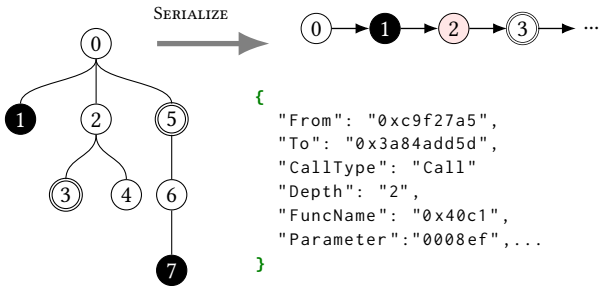


Figure 2: The serialization of a CFT, which corresponds to a raw log. ○ denotes a call node and ◎ denotes an event node.

The term *event* in this paper specifically refers to the *event* in the context of semantic log.

3.1 Stage 1: Transaction Tracing

Transaction Execution. Our target system for monitoring is the Ethereum blockchain, where numerous of transactions occur every second, each of which generates a transaction log. A raw transaction log is generated from the execution of an external transaction, which may subsequently lead to several internal transactions. Following the method described in [16], we instrument logging code into the EVM and then execute Ethereum transactions with the instrumented EVM to obtain transaction logs. The raw transaction log consists of a list of function call invocations and events. We introduce the Call Flow Tree (CFT) [16] to capture this information. The CFT specifies how the function calls are invoked and events are triggered as exemplified in Figure. 2.

There are two types of nodes in a CFT: (i) call nodes representing external and internal transactions, and (ii) event nodes representing events emitted within the transaction. Each call node includes the address of the transaction caller and callee, as well as the data carried by the transaction, which specifies the invoked function and parameters, the value of the transaction, and the type of call (e.g., `CALL`, `CALLCODE`, and `CREATE`). Each event node includes the address of the initiator and the event data. In a CFT, a directed edge between two nodes indicates that the parent node derives the child node. If a node has multiple children, we stipulate that the children are triggered in left-to-right order. Additionally, the triggering is depth-first: once an internal transaction is triggered, it will continue to trigger sequentially until a leaf node is reached.

CFT Serialization. A CFT provides a structured organization of the transactions, i.e., with function call and event as basic units. To obtain the transaction sequences, we serialize a raw transaction log (captured by a list of CFTs) into sequences. We employ a Depth-First Search (DFS) traversal algorithm for serialization to ensure that the order of each unit within the sequence mirrors the order of execution, i.e., function calls and corresponding events are processed as they appear in a log. DFS explores all nested calls before returning to the parent context, thereby preserving the linear order of events and reflecting the transaction's execution flow accurately.

After serializing all of the CFTs and concatenating them, we get a stream $s = \langle c_0, c_1, \cdots, c_n \rangle$, where c_i is the dictionary data associated with a function call or event within a transaction. The

stream s is what we called *semantic log*, which is taken as input for semantic lifting (stage 2). This refined representation offers a detailed insight into system behaviors, facilitating the extraction of semantic events.

3.2 Stage 2: Semantic Lifting for Monitoring

To determine whether a transaction sequence satisfies a MFOTL property (which characterizes a kind of attack as we will discuss later), we need to bridge the gap between system logs in the form of data sequence and high-level *events* that are referred to in the specficiation of MFOTL properties. In the following, we model the log system formally, giving its syntax and corresponding semantics, which is applied to derive the events used for monitoring from the system logs.

Syntax. The system log can be specified using a Domain-Specific-Language defined in BNF shown below:

$$\textbf{Dict } c ::= \{(k : v)^*\}$$
$$\textbf{key } k ::= k_t \mid k_e$$
$$\textbf{call } k_t ::= \mathsf{From} \mid \mathsf{To} \mid \mathsf{Func} \mid \mathsf{Depth}$$
$$\textbf{event } k_e ::= \mathsf{Data} \mid \mathsf{Topics} \mid \mathsf{Events}$$
$$\textbf{value } v \in \mathbb{N} \cup \mathbb{S}$$
$$\textbf{system log } s ::= c \mid s; c.$$

Here the statement c is dictionary data presented in Figure 2, s is the system log after serializations. $v \in \mathbb{N} \cup \mathbb{S}$ is the value of k, which is extracted from c, and $\mathbb{S}$ represent a value of string type.

As illustrated in Section 2.1, two *disjoint* sets L_t and L_e are defined to represent function call and transactions events respectively, i.e., $L_{t(e)} = \{c \mid c = \{(k_{t(e)} : v)^*\}\}$. We define several projections to extract attributes from a dictionary c, e.g., p_{f} to extract function name, p_{s} and p_{r} to extract sender and receiver address, and p_{d} to extract the call depth. Taking the data c in Figure 2 as an example, we can determine the address of transaction caller which is $p_{\mathsf{s}}(c) =$ `0x9f27a5`, the depth of the transaction which is $p_{\mathsf{d}}(c) = 1$, etc.

Semantics. We model the basic semantics of the log system with a judgement of the following form:

$$\Sigma \vdash \Pi \xrightarrow{c} \Pi',$$

which specfies a state transition. Here Σ is the static context, e.g., the set of valid addresses for EOAs and CAs, and Π and Π' are sequences of event set, which can be translated into semantic logs that can be readily monitored and verified using existing runtime verification engines. Intuitively, the above semantics states that: *given system environment* Σ *and semantic log* Π*, the execution of the atomic statement* c *yields the semantic log* Π'. The concrete semantics is defined in Figure 3, which also uses derivation rules to obtain complete semantic logs from given input. The Basic rule

$$\frac{valid(\Sigma, c),\ \ \Pi = \langle \pi_1, ...\pi_n \rangle}{\Sigma \vdash \Pi \xrightarrow{c} \Pi{+}{+}\mathcal{T}(c)}$$

specifies the state transition caused by atomic statement (a single function call or an event) in our DSL. Namely, given a statement c, the execution of c produces one or more semantic *events* which extend the semantic log Π to $\Pi{+}{+}\mathcal{T}(c)$. The operator ++ denotes concatenation. $\mathcal{T}$ is a map that maps c to a set of *observable events* presented in Table 1. The definition of $\mathcal{T}$ (shown in Figure 4) enables the extraction of different basic events from a dictionary data. As notations, we write $\{k : v\} \subseteq c$ to indicate that the dictionary c has the attribute k with the value v. The complete BTS used in this work is presented in Table 7.

Table 1: The signatures of BTS used in semantic logs.

Signature	Description
Depth(d:int)	*The function is called at depth* d.
Order(o:int)	*Represents the sequence of current call throughout the transaction*
Call(sdr,rcv,func)	*Account* sdr *calls function* func *of account* rcv.
Transfer(sdr, rcv, ast, amt)	*Account* sdr *initiates a transfer that transfers* amt *amount of* ast *asset to account* rcv.
Generate(sdr, rcv, ast, amt)	*A token contract* sdr *mints the amount* amt *of the asset* ast .
Destroy(sdr, rcv, ast, amt)	*Account* sdr *burns the amount* amt *of the asset* ast.

Basic Transaction Semantics (BTS). We take three basic events as examples to illustrate the extraction of basic events.

Call. Performing function calls is essential for any account to execute flexible operations. If c corresponds to a transaction, i.e., $c \in L_t$, we can extract a basic event `Call(sdr, rcv,func)`, where `sdr` is the account of operator, `rcv` is the account to be called, and `func` refers to the hash value of the called function. Besides, both the depth and the call index are recorded with events `Depth(d)` and `Order(o)`, where attributes `d` and `o` can also be extracted.

Transfer. The action of transferring assets between accounts can be extracted from the data c corresponding to an transaction event, i.e., $c \in L_e$, using the event `Transfer(sdr, rcv, ast, amt)`, where `sdr` is the address of the token sender, and `rcv` is the address of the token receiver. Attribute `ast` represents the type of token (assets) to be transferred and `amt` represents the transfer amount.

Generate can be viewed as a specific instantiation of Transfer that characterizes a particular type of money transfer behavior. Consider the event `Transfer(0x00, rcv, ast, amt)`, where the sender address is `0x00`, indicating that all tokens are transferred from zero addresses, thereby representing token generation. Thus we can rewrite this as a new predicate `Generate(rcv, ast, amt)`.

Example 1. Applying the function $\mathcal{T}$ to the data c in Figure 2, we obtain a set of basic events, where c corresponds to a function call, which is called at index 3 and depth 2. With projections defined above and applied rule (1), (2), and (3) in Figure 4, we can obtain that $\mathcal{T}(c) = \{\mathsf{Depth}(2), \mathsf{Order}(3), \mathsf{Call}(\texttt{0xc9}, \texttt{0x3a}, \texttt{0x40})\}$.

Advanced Attack Semantics (AAS). In addition to those basic events constructed from the statements using the BTS, we define a set of rules that generate auxiliary events that are necessary for the MOFTL monitoring of attacks. Intuitively, the rule

$$\frac{\Pi = \langle \pi_1, ...\pi_n \rangle, \psi(\pi_i, \pi_j), \Pi' = \Pi\{i \mapsto \pi_i \cup \{\rho_\psi(\pi_i, \pi_j)\}\}}{\Sigma \vdash \Pi \rightsquigarrow \Pi'}$$

states that: *given system environment* Σ *and semantic log* Π*, if there exists event sets* π_i *and* π_j $(i < j \leq n)$ *in* Π*, that satisfy the condition*

[Basic]

$$\frac{valid(\Sigma, c),\ \ \Pi = \langle \pi_1, ...\pi_n \rangle}{\Sigma \vdash \Pi \xrightarrow{c} \Pi{+}{+}\mathcal{T}(c)}$$

[Seq]

$$\frac{\Sigma \vdash \Pi \xrightarrow{s} \Pi_1, \Sigma \vdash \Pi_1 \xrightarrow{c} \Pi',}{\Sigma \vdash \Pi \xrightarrow{s;c} \Pi'}$$

[Lift]

$$\frac{\Pi = \langle \pi_1, ...\pi_n \rangle, \exists i.\varphi(\pi_i, \pi_{i+1}), \Pi' = \Pi\{i \mapsto \pi_i \cup \{\rho(\pi_i, \pi_{i+1})\}\}}{\Sigma \vdash \Pi \rightsquigarrow \Pi'}$$

[Comp]

$$\frac{\Sigma \vdash \Pi_1 \rightsquigarrow \Pi_2, \Sigma \vdash \Pi_2 \rightsquigarrow \Pi_3, ... \Sigma \vdash \Pi_{n-1} \rightsquigarrow \Pi_n}{\Sigma \vdash \Pi_1 \rightsquigarrow_* \Pi_n}$$

[Eval]

$$\frac{\Sigma \vdash \Pi_1 \xrightarrow{s} \Pi_2, \Sigma \vdash \Pi_2 \rightsquigarrow_* \Pi_n,}{\Sigma \vdash \Pi_1 \overset{s}{\Rightarrow} \Pi_n}$$

Figure 3: Selection of operational semantics

$$\mathcal{T}(c) = \begin{cases} \mathsf{Depth(d)}, & \textbf{if } \{\texttt{"Depth"} : \mathsf{d}\} \subseteq c \\ \mathsf{Order(i)}, & \textbf{if } \{\texttt{"Order"} : \mathsf{i}\} \subseteq c \\ \mathsf{Call(s, r, func)}, & \textbf{if } c \in L_t \\ \mathsf{Transfer(s, r, ast, amt)}, & \textbf{if } c \in L_e, p_{\mathsf{amt}}(c) \neq 0 \\ \mathsf{Generate(r, ast, amt)}, & \textbf{if } c \in L_e, p_{\mathsf{amt}}(c) = 0 \end{cases}$$

Figure 4: The definition of $\mathcal{T}$. Here p_{amt} is a projection that extract the value of attribute `amount` from a dictionary.

$\psi(\pi_i, \pi_j)$, then the semantic log will be updated to Π' where an event $\rho_\psi(\pi_i, \pi_j)$ will be added to π_i. The detailed definitions of ψ and corresponding ρ_ψ are placed in Appendix C.

Here we explain the semantics of two advanced attack events and show the extraction of them with Lift rule. The full explanation can be found in Table 8 in Appendix B.

SameCall. One significant behavior of the Reentrancy attack is repeatedly entering the contract, i.e., calling the same function to get a profit. We have recorded function calls in semantic logs at every time-step. If a function is called consecutively (with call index `i` and `j` respectively) with the same `sdr` and `rcv`, i.e., `Call(sdr, rcv,func)` occurs in both π_i and π_j, then we add a new event `SameCall(func, i,j)` to the event set π_i, which indicates that `func` is called twice.

Transact. We define a `Transact` action to represent an account transferring some amount of one asset, followed by a vault contract or DEX pool transferring another asset. The action can be described with predicate Transfer earlier in this section. Consider two different events: `Transfer(sdr, rcv, ast, amt)`, and `Transfer(rcv, sdr, ast', amt')`, where the attributes `ast` and `ast'` denote two different kinds of assets, satisfying `ast` $\neq$ `ast'`. We can observe that the the address of sender and receiver within two transfers are exchanged, which indicates a "transact" illustrated above. We use the predicate `Transact(opr, pool, astIn, astOut, amtIn, amtOut)` to denote the semantics of the combination of such two transfer events. Here the lifting event, which should be added in π_i should satisfy `opr = sdr, pool = rcv, astIn = ast, astOut = ast'` and `amtIn,amtout = amt,amt'`.

Our top-level judgment about semantics is of the form:

$$\Sigma \vdash \Pi \overset{s}{\Rightarrow} \Pi'.$$

Here the relation $\Rightarrow$ includes both basic events extraction ($\Sigma \vdash \Pi \rightarrow \Pi'$) and semantic lifting ($\Sigma \vdash \Pi \rightsquigarrow \Pi'$), s is the system log defined in above DSL syntax, and Π' is the final semantic log, which cannot be lifted further. Given a system log s and an empty list, the semantic log Π used for monitoring satisfies $\Sigma \vdash \langle\rangle \overset{s}{\Rightarrow} \Pi$.

3.3 Stage 3: Attack Formulas and Detection

We capture transaction-level attacks using MOFTL formulas, which allows us to reuse existing runtime monitoring frameworks such as MonPoly to detect attacks based on the semantic logs.

In the following, we consider two significant transaction-level attacks which have occurred and resulted in substantial losses on blockchain system as examples, and we show how to characterize the complicated behavior of adversary with MFOTL. For each attack, we first summarize the basic workflow of the attack, from which we identify the core signature and logical relationships. Then we illustrate each attack formulation in detail. The remaining attacks are described in Appendix D.

Reentrancy (RE). At its core, a Reentrancy attack leverages the asynchronous nature of smart contracts to manipulate their behavior in unintended ways. Smart contracts on blockchains like Ethereum operate in a deterministic and sequential manner, i.e., once a function is called within a contract, it must complete execution before another function can be invoked. However, in the case of a Reentrancy attack, an attacker exploits this by recursively calling the same function within the contract before the previous invocation completes. We can define the attack as follows:

```
1 Reentrancy_Attack
2 let OrderDepth(o,d):= Order(o) ∧ Depth(d) in
3 let Func(s,r,f,o,d):= OrderDepth(o,d) ∧ Call(s,r,f) in
4 let Inv(o,o',d'):= Depth(d') ∧ InverseCall(o,o') in
5 let SameFunc(f,o,o',d'):= Depth(d') ∧ SameCall(f,o,o') in
6 let NestedDepth(x,y,z):= x > y ∧ y > z in
7     Func(s1,r1,f,o1,d1) ∧ ◇I Inv(o1,o3,d3) ∧
8     ◇I SameFunc(f,o1,o2,d2) ∧ NestedDepth(d1,d2,d3)
```

We use MFOTL's `let-in` operator to define predicates for convenience (Line 2-6). The given MFOTL formula describes Reentrancy attacks on Ethereum. Initially, the attacker $\mathtt{s}_1$ calls function `f` in the victim contract $\mathtt{r}_1$ with depth $\mathtt{d}_1$, represented by the predicate `Func` in Line 7. The victim contract then makes a callback to the attacker's contract, captured by $\Diamond_I$`Inv` in Line 7. The attacker reenters the victim contract and invokes the same function again, as indicated by the predicate `SameFunc` in Line 8. These calls are nested, with increasing depth, as indicated by the `NestedDepth` predicate in Line 8. The `OrderDepth` predicate (Line 2) denotes their indices (both depth and order) in the semantic log. The interval I is empirically chosen. The formula describes the attacker entering the victim contract twice, exploiting the failure to update its state promptly by withdrawing funds before the state refreshes during external calls.

Direct Price Manipulation (DPM). DPM attacks pose a significant threat to the stability and integrity of DeFi ecosystems on

blockchains. The mechanism of a DPM attack typically involves the manipulation of the price oracle used by the decentralized exchange or automated market maker. Oracles are external sources of data that provide price information for assets traded on the blockchain. In a DPM attack, an attacker can manipulate the price reported by the oracle by providing false or misleading data. Once the price reported by the oracle is manipulated, the attacker can exploit this discrepancy to execute trades at advantageous prices on the decentralized exchange or automated market maker. By buying or selling large volumes of assets at artificially inflated or deflated prices, the attacker can influence the market and potentially profit from the price movements.

```
DPM_Attack
let Tr(o,op,p,ast,ast',amt,amt'):= Order(o) ∧
    Transact(op,p,ast,ast',amt,amt') in
let SamePool(x,y,z):= x = y ∧ y = z in
    Tr(o₂,op₂,p₂,ast₂,ast₂',amt₂,amt₂') ∧
    ♦_I Tr(o₁,op₁,p₁,ast₁,ast₁',amt₁,amt₁') ∧
    ◇_I Tr(o₃,op₃,p₃,ast₃,ast₃',amt₃,amt₃') ∧
    SamePool(p₁,p₂,p₃) ∧  op₁ = op₃ ∧
    ast₁ = ast₃' ∧ ast₃ = ast₁' ∧ ast₁'= ast₂'
```

The given formula describes DPM attack. The victim $\mathtt{op}_2$ executes a token swap, exchanging asset $\mathtt{ast}_2$ for asset $\mathtt{ast}_2$'. This action is represented by the first Tr predicate. However, before this transaction, the attacker $\mathtt{op}_1$ has already swapped $\mathtt{ast}_1$ for $\mathtt{ast}_1$', where $\mathtt{ast}_1$' is the same with $\mathtt{ast}_2$', as captured by $\blacklozenge_I$Tr predicate. This swap impacts the liquidity pool's price oracle, causing the price of the asset to rise rapidly, resulting in the victim receiving fewer $\mathtt{ast}_2$' assets than anticipated. Subsequently, the attacker reverses the initial swap, captured by $\Diamond_I$Tr predicate. Furthermore, all transactions occur within the same pool, described by SamePool predicate.

Monitoring with MonPoly. With attack formulas φ specified in this section, we can use MonPoly to monitor the semantic log Π obtained from stage 2 to detect that if is raised an attack. The tool MonPoly outputs a set of predicates and their parameters that match the attack formulas. Each predicate and its associated parameters indicate where the attack formulas φ are satisfied within the semantic log, i.e., where the attack occurs and with what parameter values. In Section 4, the effectiveness, efficiency, and the precision of our detection approach will be evaluated in detail.

3.4 Quantitative Analysis

Due to the flexibility that the adversary has when carrying out attacks, some attacks closely resemble the attack formula but fail to fully satisfy it. There are also near-miss attacks that attempt but ultimately fail to succeed. In addition to qualitatively determining whether an attack happens (the attack formula is satisfied under the semantics of MFOTL), we further equip MoE with a quantitative analysis capability, i.e., to *measure the similarity between a semantic log and a valuation of an attack*. The motivation behind this are twofolds: 1) to improve the attack detection performance by computing the similarity valuation trends of a transaction; 2) to provide an opportunity for the early alarming the occurrence of an attack. In the following, we first recall the syntax of our attack formula then define the quantitative semantics.

Note that for the attack modeling in this work, it is sufficient only to use a subset of MFOTL specification language. The minimal syntax of attack formula is in Figure 8, where φ_{atm} is an atomic formula, which is either an event $e(\bar{t})$ or $\text{op}(t,t)$. The operator op includes (in)equality =, less than < and other binary operators. The temporal operators $\Diamond_I$ and $\blacklozenge_I$ denotes "eventually" and "once" respectively.

Definition 1 (Quantitative Semantics). *Given an MFOTL formula φ and an event set π_i within a semantic log Π, the quantitative semantics can be evaluated using a function ϱ, which is defined recursively as follows:*

$$\varrho(\varphi_{\text{atm}}, i, \Pi) = c_a, \text{ if } \exists v.\ v,\ i \models_\Pi \varphi_{\text{atm}} \text{ and } \varphi_{\text{atm}} \in \Phi'_a, \quad \textbf{(T1)}$$

$$\varrho(\varphi_{\text{atm}}, i, \Pi) = c_b, \text{ if } \exists v.\ v,\ i \models_\Pi \varphi_{\text{atm}} \text{ and } \varphi_{\text{atm}} \in \Phi'_b, \quad \textbf{(T2)}$$

$$\varrho(\varphi_{\text{atm}}, i, \Pi) = 0, \text{ if } \forall v.\ v,\ i \not\models_\Pi \varphi_{\text{atm}}. \quad \textbf{(T3)}$$

$$\varrho(\neg\varphi_{\text{atm}}, i, \Pi) = c_a - \varrho(\varphi_{\text{atm}}, i, \Pi), \text{ if } \varphi_{\text{atm}} \in \Phi'_a \quad \textbf{(T4)}$$

$$\varrho(\neg\varphi_{\text{atm}}, i, \Pi) = c_b - \varrho(\varphi_{\text{atm}}, i, \Pi), \text{ if } \varphi_{\text{atm}} \in \Phi'_b \quad \textbf{(T5)}$$

$$\varrho(\Diamond_I \varphi_{\text{atm}}, i, \Pi) = \max_{j \text{ s.t. } \tau_j - \tau_i < I} \left\{ \varrho(\varphi_{\text{atm}}, j, \Pi) \cdot \frac{|I| - (\tau_j - \tau_i)}{|I|} \right\}, \quad \textbf{(T6)}$$

$$\varrho(\blacklozenge_I \varphi_{\text{atm}}, i, \Pi) = \max_{j \text{ s.t. } \tau_i - \tau_j < I} \left\{ \varrho(\varphi_{\text{atm}}, j, \Pi) \cdot \frac{|I| - (\tau_i - \tau_j)}{|I|} \right\}, \quad \textbf{(T7)}$$

$$\varrho(\varphi_1 \wedge \varphi_2, i, \Pi) = \varrho(\varphi_1, i, \Pi) + \varrho(\varphi_2, i, \Pi). \quad \textbf{(T8)}$$

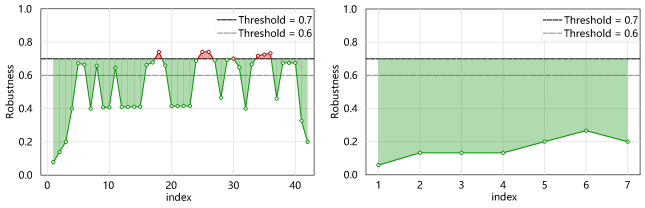
Here, the relation $v, i \models_\Pi \varphi$ denotes the satisfaction of the formula φ for a valuation v at an index i with respect to the trace Π. The constants c_a and c_b are precomputed for a given attack formula φ. $\varphi \in \Phi'_a$ denotes that φ is an AAS-related formula. The value of the function ϱ consistently falls within [0, 1] (proof in Appendix E).

Intuitively, the function ϱ is defined so that *the satisfaction of each subformula contributes a higher likelihood to the satisfaction of the overall formula* (**T8** in Definition 1). Rule **T1** and **T2** state that we assign a weight of c_a to each sub-formula in Φ'_a, and a weight of c_b to each sub-formula in Φ'_b. The intuition for $c_a > c_b$ is that: the satisfaction of AAS-related atomic formulas makes a larger contribution than BTS-related atomic formulas to the overall satisfaction. If π_i dissatisfies φ_{atm}, the result is 0 (**T3**). For formula involving temporal operator $\Diamond_I$ and $\blacklozenge_I$, we define **T6** and **T7**. Our intuition is that the *faster* φ_{atm} is satisfied within the interval I, the greater its contribution to the satisfaction of the overall formula φ. This is because the Ethereum's evolving state and reliance on current resource conditions can alter contract states or token prices over time, impacting the success of prolonged attacks. Besides, to make the quantitative algorithm complete, we add a definition for $\neg$ operator (**T4** and **T5**). The derivation and corresponding proof for the above definition can be found in Appendix E. A simple example of quantitative analysis is provided in Appendix F.

Definition 2 (Attack Alarm). Given a semantic log $\Pi = \langle \pi_1, \ldots \pi_n \rangle$, an attack formula φ and a threshold τ_c, an attack alarm is raised at index i, when $\varrho(\varphi, i, \Pi) > \tau_c$ holds.

With the condition of an attack alarm occurrence defined above, we can report some potential attacks which may not be detected by direct monitoring with MonPoly. We will conduct several case studies in Section 4.4, and here we only give a simple example.

Example 2. To intuitively showing this effect of quantitative analysis, we evaluate the quantitative results of an IPM attack and the



(a) Evaluation of an IPM. **(b) A Normal Transaction.**

Figure 5: Quantitative analysis for IPM (Example 2).

normal transaction, as shown in Figure 5. Red points denote robustness exceeding the threshold $\tau_c = 0.7$, while green areas indicate no attacks. Figure 5a highlights attack instances with robustness over the threshold, occurring mid-transaction. Figure 5b shows a normal transaction with values below the threshold, indicating no detected attacks. Lowering the threshold to 0.6 detects attacks earlier but risks more false positives. Attacks often occur before a transaction ends, and the threshold adjustment affects attack identification. This underscores the importance of quantitative semantic monitoring for detecting various Ethereum attack patterns.

4 Experimental Evaluation

In this section, we evaluate the detection capability of MoE by answering the following research questions (RQs):
RQ1: Can MoE detect attack transactions in real-world incidents?
RQ2: Can MoE analyze large-scale transactions?
RQ3: Can MoE detect more attacks with its quantitative capability?
RQ4: Can MoE efficiently reveal attacks?

4.1 Experimental Setup

Dataset. Two datasets, **D1** and **D2**, listed in Table 2, are built to comprehensively evaluate the effectiveness of MoE . **D1** is a labeled benchmark constructed by expanding an existing dataset [16] with real-world incidents from May 2020 to June 2022, covering 24 documented attacks. This benchmark is essential for validating the accuracy of MoE in detecting known attack patterns based on publicly reported security incidents. **D2**, is collected from 59 high-volume Dapps listed on DeFiLlama [7] (e.g., Value Defi, Uniswap, RaRi, etc.) and includes 1,064,996 unlabeled transactions from May 2020 to June 2022 (blocks 10,000,000 to 15,000,000). This large-scale dataset helps assess MoE 's performance, efficiency, and accuracy in detecting attacks in a real-world, unlabelled environment. All transaction data can be accessed publicly at [2].

Implementation. For *transaction tracing*, we insert recording code in a off-chain transaction execution environment [11] and use it to execute the transactions to obtain system logs. For *Semantic Lifting*, we implement a parser that convert system logs to the semantic logs for runtime verification. For *runtime verification*, we modify MonPoly so that it supports our quantitative analysis, and run the modified MonPoly with the specified attack formulas and semantic logs to detect attack transactions.

4.2 Detection in Labeled Benchmark (RQ1)

In this experiment, we evaluate the effectiveness of MoE by deploying it to monitor each type of incident in the labeled benchmark **D1**. MoE tags the transactions that satisfy the specific attack formula.

Table 2: Datasets.

ID	Source	Num. of Tx	Used in RQs
D1	24 security incidents	24	RQ1, RQ3
D2	59 well-known Dapps	1,064,996	RQ2, RQ3, RQ4

Table 3: Results in labeled benchmark.

Attack Type	Num.	DeFiWarder		MoE	
		#TP	TRP	#TP	TRP
Sandwich	5	/	/	5	1.00
Reentrancy	8	4	0.50	7	0.88
CI	2	1	0.50	2	1.00
DPM	3	2	0.67	3	1.00
IPM	6	2	0.33	5	0.83
ALL	24	9	0.47	22	0.92

Table 4: Results in large-scale experimantal evaluation. The symbol * denotes that the value is estimated.

Attack Type	Tagged Transaction	TP	Precision
Sandwich	2,433	2,433*	1.00
Reentrancy	213	181*	0.85
CI	761	647*	0.84
DPM	18	15	0.83
IPM	53	43	0.81
All	3,478	3,319*	0.95

We mark the tagged transaction as true positive (TP), and mark the attack as false negative (FN) if it is not tagged. We evaluate the effectiveness of MoE with the recall rate: $TRP = \frac{TP}{TP+FN}$.

Evaluation of experimental results. The results on the labeled benchmark are presented in Table 3. MoE shows excellent recall for Sandwich, Call Injection and Direct Price Manipulation attacks, covering all labeled transactions. This indicates that our MFOTL formulas effectively adapt to real-world attack behaviors. However, MoE fails to detect 2 attacks: one Reentrancy attack, where the attacker exploits a vulnerable function while profiting from another, and one Indirect Price Manipulation attack, which uses special operation pairs to manipulate token prices. Addressing these requires more behavioral information, but the overhead is considerable. Overall, MoE successfully reveals 92% ($\frac{22}{24}$) of attacks in the benchmark, demonstrating its effectiveness in detection.

Comparison with other tools. We adapt DeFiWarder [16] for our dataset, showing that MoE has a 45% higher recall rate in Table 3. In particular, DeFiWarder cannot detect Sandwich attacks since its abnormal detection cannot distinguish attack and benign arbitrage. Furthermore, DeFiWarder relies on a large number of historical transactions to obtain a normal return rate for abnormal detection, which limits its usability. In contrast, MoE can detect attacks without learning from historical transactions.

4.3 Detection in Massive Transactions (RQ2)

To evaluate the practicality of MoE , we use it to detect attack transactions in the large-scale dataset **D2**.

Evaluation of experimental results. MoE tags 3,478 transactions as attacks from 1,064,996 transactions. We evaluate these tagged transactions $Precision = \frac{TP}{TP+FP}$, Specifically, we manually distinguish TP and FP in transactions. For SW, RE and CI, we sample 332, 138 and 256 transactions, respectively, for statistical significance at 95% confidence level. We calculate precision and labeled as "Estimated" in Table 4. All Direct/Indirect Price Manipulation transactions are verified independently by two authors.

Table 4 shows the experimental results. MoE detects 2,433 Sandwich attacks with 100% precision and 213 Reentrancy transactions with an acceptable precision rate (85%). Some false positives occurred because repeated queries in lending operations mimicked Reentrancy behavior, although these were read-only operations. MoE has 84% precision in detecting Call Injection attacks. Among *FP*, some of the transactions executed multiple transfer operations in a call. However, these transfer operations were batch processed through the 'multiSend' method. As a result, multiple 'Transfer' events were triggered, but this does not constitute an attack. For Direct and Indirect Price Manipulate attacks, MoE identifies 15 direct and 43 indirect attacks with over 81% precision, though some arbitrage operations were misclassified.

4.4 Quantitative Analysis using MoE (RQ3)

We apply MoE to quantify the similarity of transactions with attacks using our quantitative semantics. We use **D1** for positive samples and randomly select 20 normals from **D2** as negatives. To assess the impact of threshold τ_c, we run MoE with τ_c at 0.6, 0.7, 0.8, and MoE without quantitative analysis. These thresholds are based on the quantified semantic like "Eventually" and "Once", where increasing distance leads to decreased robustness, preventing even perfectly matching attacks from achieving a robustness of 1.0. We select a threshold of 0.8 to capture transactions with distinct attack characteristics. Lowering the threshold can identify more attacks, including near-miss attacks, but may misclassify normal transactions. Thus, we also test thresholds of 0.6, 0.7 and 0.8 to evaluate the trade-off between detecting more attacks and maintaining accuracy.

Evaluation of experimental results. Our experimental results using the MoE monitoring tool are shown in Table 5. Higher TP and lower FN values are ideal, as they indicate more correct detections of positive samples and fewer tagged negative samples, respectively.

Data shows each attack type needs unique threshold tuning for optimal performance. All attack types can tag all positives at certain thresholds (indicated by *), but may increase undesirable FNs. RE is fully tagged at $\tau_c = 0.7$ with 2 FPs. SW and CI are detected at $\tau_c = 0.8$ without any FPs. With our quantitative semantics, MoE detects 8% more true attacks than non-quantitative version, covering all incidents in benchmark. Fortunately, our quantitative analysis enhances MoE 's attack detection with an acceptable FP rate.

Importantly, the use of our quantitative semantics allowed MoE to detect a near-miss attack on Warp Finance[15], which the non-quantitative version missed. In this case, the attacker manipulated the price of LP tokens and held them as collateral, but Warp Finance locked these LP tokens, preventing any profit. Although the attack was not fully executed, the quantitative analysis enabled MoE to identify the attack behavior and trends, allowing it to classify the incident as an attack.

Table 5: Threshold Evaluation Results: '*' denotes all positive samples; 'w/o' in the Threshold column indicates detection results from MoE without quantitative analysis.

Attack	Threshold	TP	FP	Precision	Recall	Accuracy
DPM	0.6	3*	1	0.75	1.00	0.96
	0.7	3*	1	0.75	1.00	0.96
	0.8	0	0	/	0	0.87
	w/o	3*	0	1.00	1.00	1.00
IPM	0.6	6*	1	0.86	1.00	0.96
	0.7	6*	1	0.86	1.00	0.96
	0.8	1	0	1.00	0.17	0.81
	w/o	5	0	1.00	0.83	0.96
CI	0.6	2*	1	0.67	1.00	0.95
	0.7	2*	1	0.67	1.00	0.95
	0.8	2*	0	1.00	1.00	1.00
	w/o	2*	0	1.00	1.00	1.00
RE	0.6	8*	6	0.57	1.00	0.79
	0.7	8*	2	0.80	1.00	0.93
	0.8	7	0	1.00	0.88	0.96
	w/o	7	0	1.00	0.88	0.96
SW	0.6	5*	2	0.71	1.00	0.92
	0.7	5*	2	0.71	1.00	0.92
	0.8	5*	0	1.00	1.00	1.00
	w/o	5*	0	1.00	1.00	1.00

4.5 Efficiency of MoE (RQ4)

We evaluate MoE 's run-time verification efficiency on 1,064,996 transactions, logging the average time per transaction for stages including *transaction tracing*, *semantic lifting*, *attack verification*, and their total consumption time. The time of *attack verification* varies due to the different formulas. Our experimental results are given in the Appendix, Table 9.

MoE processes *transaction tracing* in 5.20 ms, with minimal overhead over EVM (4.78 ms). *Semantic lifting* takes just 0.13 ms/tx, the quickest of the three stages. Finally, *attack verification* shows Sandwich detection has the highest efficiency at 6.45 ms/tx or 155 TPS due to its simple formula. Other attacks, like DPM take longer (23.43 ms/tx or 42.68 TPS) due to the formula's complexity. MoE 's TPS exceeds Ethereum's 13.4 TPS [8], indicating that MoE is capable of real-time monitoring.

5 Conclusion

In this work, we introduced MoE, a flexible and extensible runtime monitoring framework specifically designed for detecting a variety of transaction-level attacks on the Ethereum blockchain. Leveraging the expressive capabilities of MFOTL and our proposed novel semantic lifting approach, MoE effectively formalizes and identifies both known and new attack types.

Our approach, integrated with the state-of-the-art runtime monitoring tool MonPoly enhanced with quantitative analysis, demonstrates high efficiency in processing large-scale Ethereum transaction logs. In an evaluation of over one million transactions from 59 Ethereum Dapps, MoE successfully detected various attack types, including sandwich attacks, reentrancy attacks, and price manipulation attacks, all while maintaining low overhead. These results highlight the framework's scalability and its potential application in proactive runtime monitoring for blockchain security.

References

[1] Aicoin. 2023. Information about Ethereum. https://www.aicoin.com/currencies/ethereum.html?lang=en.

[2] Anonymous. 2024. https://anonymous.4open.science/r/Ethereum-Transaction-C029/.

[3] David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. 2012. MONPOLY: Monitoring usage-control policies. In *Runtime Verification: Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers 2*. Springer, 360–364.

[4] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. 2015. Monitoring metric first-order temporal properties. *Journal of the ACM (JACM)* 62, 2 (2015), 1–45.

[5] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–43.

[6] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 910–927.

[7] DeFillama. 2023. DeFillama Dashboard. https://defillama.com/..

[8] Etherscan. [n. d.]. Etherscan Blockchain Explorer. https://etherscan.io/

[9] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 259–269.

[10] Ali Safavi Kevin Wang. 2016. Blockchain is empowering the future of insurance. https://techcrunch.com/2016/10/29/blockchain-is-empowering-the-future-of-insurance/.

[11] Yeonsoo Kim, Seongho Jeong, Kamil Jezek, Bernd Burgstaller, and Bernhard Scholz. 2021. An Off-The-Chain Execution Environment for Scalable Testing and Profiling of Smart Contracts.. In *USENIX Annual Technical Conference*. 565–579.

[12] Daniel Perez and Benjamin Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1325–1341. https://www.usenix.org/conference/usenixsecurity21/presentation/perez

[13] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 198–214.

[14] Rekt. 2020. VALUE DEFI - REKT. https://rekt.news/value-defi-rekt/.

[15] Rekt. 2020. WARP FINANCE - REKT. https://rekt.news/zh/warp-finance-rekt/.

[16] Jianzhong Su, Xingwei Lin, Zhiyuan Fang, Zhirong Zhu, Jiachi Chen, Zibin Zheng, Wei Lv, and Jiashui Wang. 2023. DeFiWarder: Protecting DeFi Apps from Token Leaking Vulnerabilities. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1664–1675. https://doi.org/10.1109/ASE56229.2023.00110

[17] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing, and Baoxu Liu. 2021. Evil under the sun: understanding and discovering attacks on Ethereum decentralized applications. In *30th USENIX Security Symposium (USENIX Security 21)*. 1307–1324.

[18] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–38.

[19] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. 2020. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering* 8, 2 (2020), 1133–1144.

[20] Zikai Alex Wen and Andrew Miller. 2016. Scanning Live Ethereum Contracts for the "Unchecked-Send" Bug. https://rekt.news/zh/warp-finance-rekt/.

[21] Siwei Wu, Zhou Yu, Dabao Wang, Yajin Zhou, Lei Wu, Haoyu Wang, and Xingliang Yuan. 2023. DeFiRanger: Detecting DeFi Price Manipulation Attacks. *IEEE Transactions on Dependable and Secure Computing* (2023), 1–15. https://doi.org/10.1109/TDSC.2023.3346888

[22] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. 2017. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE international congress on big data (BigData congress)*. Ieee, 557–564.

[23] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. 2020. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems* 105 (2020), 475–491.

[24] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. 2018. Blockchain challenges and opportunities: A survey. *International journal of web and grid services* 14, 4 (2018), 352–375.

[25] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. 2021. High-frequency trading on decentralized on-chain exchanges. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 428–445.

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

[27] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2021. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 3283–3290.

A An example of Price Manipulation Attack

Figure 6 illustrates a high-level example of a price manipulation attack. This attack involves a series of complex transactions across multiple DeFi protocols to manipulate asset prices for financial gain. Specifically, the attacker initiated the attack by swapping 116M DAI and 31M USDT for 107.5M USDC, which significantly impacted the pricing mechanism of Curve. As shown in Table 6, this disruption allowed the attacker to acquire 31M 3CRV tokens, a higher amount than expected, as proof of deposited assets. In the final step, the attacker swapped the 3CRV tokens back for more DAI than initially deposited, ultimately profiting 8M DAI. The figure provides a visual breakdown of these steps, while the table details the asset balance changes during each stage of the attack. The details of major transaction-level attacks are presented in Sect. 3.3.

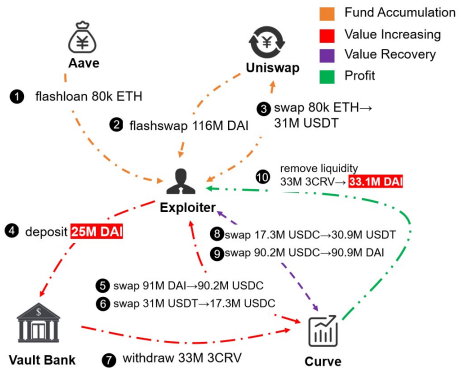


Figure 6: A Price Manipulation Attack on Value Defi.

Table 6: The balance of exploiter in value Defi attack.

	DAI	USDT	USDC	3CRV
fund accumulation	**116M**	31M	0	0
value increasing	0	0	107.5M (90.2M+17.3M)	33M
value recovery	90.9M	30.9M	0	33M
profit	**124M (90.9M+33.1M)**	30.9M	0	0

B Description of BTS and AAS Events

Here we give an overall description BTS and AAS events used in this work. For each event, we give its signature and explain the intuitions for it.

The BTS events we used include: `Depth`, `Order`, `Call`, `Transfer`, `Generate`, `Destroy`, etc.

The AAS events we used include: `Samecall`, `InverseCall`, etc.

Table 7: The signatures of BTS used in semantic logs.

Signature	Description
Depth(d:int)	*The function is called at depth* d.
Order(o:int)	*Represents the sequence of current call throughout the transaction*
Call(sdr,rcv,func)	*Account* sdr *calls function* func *of account* rcv.
Transfer(sdr, rcv, ast, amt)	*Account* sdr *initiates a transfer that transfers* amt *amount of* ast *asset to account* rcv.
Generate(sdr, rcv, ast, amt)	*A token contract* sdr *mints the amount* amt *of the asset* ast .
Destroy(sdr, rcv, ast, amt)	*Account* sdr *burns the amount* amt *of the asset* ast.
Swap(astIn-astout, amtIn, amtout)	*A token swap event directly extracted from Uniswap event, exchanging* amtIn *of* astIn *to* amtOut *of* astOut.
SensitiveFunc(func)	*A function* func *which makes an ownership change or token transfer.*
Calldata(p)	*The calldata* p *of a function call.*

Table 8: The signatures of AAS used in semantic logs.

Signature	Description
SameCall(func,i,j)	*A function is called consecutively (with call index* i *and* j *respectively) with the same* sdr *and* rcv.
InverseCall(i,j)	*Represents that the direction of the call with index* j *is the reverse of the call with index* i*, meaning the sender and receiver addresses are swapped between the two calls.*
Calldata_Explode(i,j)	*The parameter of a call with call index* j *is influenced by parameter of a call with call index* i.
Mint(op,pool,sin,sout,min,mout)	*An account* op *deposits certain amount* min *of certain asset* sin *to provide liquidity in a liquidity pool* pool *of a DeFi app, which then mints certain amount* mout *of its LP token* sout *to* op.
Burn(op,pool,sin,sout,min,mout)	*An account* op *burns a certain amount* min *of certain DeFi app's LP token* sin *to redeem deposits, a pool* pool *of the app then transfers a certain amount* mout *of certain asset* sout *to* op.
Transact(op,pool,sin,sout,min,mout)	*An account* op *sells a certain amount* min *of certain asset* sin *for a certain amount* mout *of certain asset* sout *in a liquidity pool* pool *of an AMM.*

C Lifting Condition ψ and Updating Function ρ_ψ

In addition to those basic events constructed from the statements using the BTS, we define a set of rules that generate auxiliary events that are necessary for the MOFTL monitoring of attacks. Intuitively, the rule

$$\frac{\Pi = \langle \pi_1, \ldots \pi_n \rangle, \psi(\pi_i, \pi_j), \Pi' = \Pi\{i \mapsto \pi_i \cup \{\rho_\psi(\pi_i, \pi_j)\}\}}{\Sigma \vdash \Pi \rightsquigarrow \Pi'}$$

states that: *given system environment* Σ *and semantic log* Π*, if there exists event sets* π_i *and* π_j $(i < j \leq n)$ *in* Π*, that satisfy the condition* $\psi(\pi_i, \pi_j)$*, then the semantic log will be updated to* Π' *where an event* $\rho_\psi(\pi_i, \pi_j)$ *will be added to* π_i. The detailed definitions of ψ and corresponding ρ_ψ are shown in Figure 7.

D Attack Detection of 3 Types of Attacks

D.1 Call Injection (CI)

Call Injection attack is a type of smart contract security vulnerability where attackers exploit this vulnerability by injecting malicious

[SameCall]

$$\frac{\psi(\pi_i, \pi_j) := \exists e.e = \mathsf{Call}(\mathsf{s}, \mathsf{r}, \mathsf{func}) \in \pi_i \cap \pi_j}{\rho_\psi(\pi_i, \pi_j) := \mathsf{SameCall}(\mathsf{func}, i, j)}$$

[Inverse]

$$\frac{\psi(\pi_i, \pi_j) := \exists e_1, e_2.e_1 = \mathsf{Call}(\mathsf{s}, \mathsf{r}, \mathsf{func}) \in \pi_i \\ \wedge e_2 = \mathsf{Call}(\mathsf{r}, \mathsf{s}, \mathsf{func}') \in \pi_j}{\rho_\psi(\pi_i, \pi_j) := \mathsf{Inverse}(\mathsf{i}, \mathsf{j})}$$

[CalldataExplode]

$$\frac{\psi(\pi_i, \pi_j) := \exists e_1, e_2, e_3, e_4.e_1 = \mathsf{Calldata}(\mathsf{p}) \in \pi_i \\ \wedge e_2 = \mathsf{Depth}(\mathsf{d}) \in \pi_i \wedge e_3 = \mathsf{Calldata}(\mathsf{p}') \in \pi_j \\ \wedge e_4 = \mathsf{Depth}(\mathsf{d}') \in \pi_j \wedge d' > d \wedge p' \in p}{\rho_\psi(\pi_i, \pi_j) := \mathsf{CalldataExplode}(\mathsf{i}, \mathsf{j})}$$

[Transact]

$$\frac{\psi(\pi_i, \pi_j) := \exists e_1, e_2.e_1 = \mathsf{Transfer}(\mathsf{s}, \mathsf{r}, \mathsf{ast}, \mathsf{amt}) \in \pi_i \\ \wedge \mathsf{ast} \neq \mathsf{ast}' \wedge e_2 = \mathsf{Transfer}(\mathsf{r}, \mathsf{s}, \mathsf{ast}', \mathsf{amt}') \in \pi_j}{\rho_\psi(\pi_i, \pi_j) := \mathsf{Transact}(\mathsf{s}, \mathsf{r}, \mathsf{ast}, \mathsf{ast}', \mathsf{amt}, \mathsf{amt}')}$$

[Mint]

$$\frac{\psi(\pi_i, \pi_j) := \exists e_1, e_2.e_1 = \mathsf{Transfer}(\mathsf{s}, \mathsf{r}, \mathsf{ast}, \mathsf{amt}) \in \pi_i \\ \wedge \mathsf{ast} \neq \mathsf{ast}' \wedge e_2 = \mathsf{Generate}(\mathtt{0x00}, \mathsf{s}, \mathsf{ast}', \mathsf{amt}') \in \pi_j}{\rho_\psi(\pi_i, \pi_j) := \mathsf{Mint}(\mathsf{s}, \mathsf{r}, \mathsf{ast}, \mathsf{ast}', \mathsf{amt}, \mathsf{amt}')}$$

[Burn]

$$\frac{\psi(\pi_i, \pi_j) := \exists e_1, e_2.e_1 = \mathsf{Destroy}(\mathsf{s}, \mathtt{0x00}, \mathsf{ast}, \mathsf{amt}) \in \pi_i \\ \wedge \mathsf{ast} \neq \mathsf{ast}' \wedge e_2 = \mathsf{Transfer}(\mathsf{r}, \mathsf{s}, \mathsf{ast}', \mathsf{amt}') \in \pi_j}{\rho_\psi(\pi_i, \pi_j) := \mathsf{Burn}(\mathsf{s}, \mathsf{r}, \mathsf{ast}, \mathsf{ast}', \mathsf{amt}, \mathsf{amt}')}$$

Figure 7: The definition of lifting condition ψ and updating function ρ_ψ

calldata to trigger victims to execute sensitive functions related to fund transfers. Attackers send meticulously designed transactions to victim contracts, causing additional calls to be inserted into the contract's call stack during execution. These extra calls may alter the contract's state or transfer funds, resulting in financial losses or abnormal contract behavior.

```
CI_Attack
let CallFunc(o,s,r,f):= Order(o) ∧
    Call(s,r,f) in
let SensitiveCall(o,s,r,f):= CallFunc(o,s,r,f) ∧
    Sensitive_Func(o,s,r,f) in
    CallFunc(o1,s1,r1,f1) ∧ ◇I SensitiveCall(o2,s2,r2,f2) ∧
    CalldataExplode(oa,ob) ∧
    r1 = s2 ∧ oa = o1 ∧ ob = o2
```

The given MFOTL formula describes a Call Injection attack on Ethereum. Initially, the attacker s_1 initiates a call to the victim contract r_1, which is represented by CallFunc, with the call index o_1. Upon receiving the calldata from s_1, the victim contract triggers an additional sensitive call with the index o_2. The address and calldata for this additional call are controlled by the attacker s_1, meaning that all the information for the secondary call is designed within the calldata of the initial call. This behavior is captured by

the CalldataExplode predicate in the formula, indicating that the call with index o_2 is dictated by the calldata of the call with index o_1. Furthermore, the functions invoked by these additional calls modify the contract state and transfer funds, which is described by the SensitiveCall predicate in the formula. The MFOTL formula encapsulates this sequence, illustrating how an attacker can inject additional, malicious calls through carefully constructed calldata, thereby manipulating the contract to execute sensitive functions that alter the state or transfer assets. This highlights the vulnerability of contracts to CI attacks where the attacker exploits the contract's handling of calldata to achieve unauthorized operations.

D.2 Indirect Price Manipulation (IPM)

IPM is a way to manipulate asset prices within DeFi ecosystems on blockchains. IPM attack involves exploiting the price mechanism of a specific Dapp (such as a lending app) to increase the value of the attacker's collateral. This manipulation allows the attacker to mint more Liquidity Provider tokens as proof of the deposited assets, enabling them to borrow more liquid assets. By artificially inflating the value of their collateral, attackers can disrupt the balance of funds within the lending platform, causing losses to the platform and its users.

```
IPM_Attack
let Tr(o,op,p,ast,ast',amt,amt'):= Order(o) ∧
    Transact(op,p,ast,ast',amt,amt') in
let Mt(o,op,p,ast,ast',amt,amt'):= Order(o) ∧
    Mint(op,p,ast,ast',amt,amt') in
    Mt(o₂,op₂,p₂,ast₂,ast₂',amt₂,amt₂') ∧
    ♦_I Tr(o₁,op₁,p₁,ast₁,ast₁',amt₁,amt₁') ∧
    ◇_I Tr(o₃,op₃,p₃,ast₃,ast₃',amt₃,amt₃') ∧
    p₁ = p₃ ∧ ¬ (p₂ = p₁) ∧ op₁ = op₃ ∧
    ast₁ = ast₃' ∧ ast₃ = ast₁'
```

The given MFOTL formula of IPM, similar to DPM, describes the attacker op_1 initiates two opposite token swaps in the liquidity pool p_1, described by the Tr predicate. These token swaps increase the market value of the tokens being used as collateral, allowing the attacker to mint more LP tokens ast'_2 as proof of the deposited assets in another collateral pool p_2. This minting operation is represented by the Mt predicate in the formula. The formula captures how the attacker manipulates the token value through strategic swaps, thereby increasing their collateral's worth and enabling the minting of additional LP tokens. In this paper, only the MFOTL formulas that describe the most common scenarios for IPM and DPM are presented. The complete set of formulas will be provided in the source code.

D.3 Sandwich (SW)

A sandwich attack is a specific type of manipulation seen in decentralized finance (DeFi) on blockchain systems, particularly within automated market makers (AMMs) and decentralized exchanges (DEXs). In this attack, a malicious actor or bot spots a pending transaction waiting to be processed and places their own transactions both before and after the targeted transaction in the same block. The first transaction typically involves buying up a specific asset to drive up its price. The victim's transaction then executes at this inflated price. Immediately afterward, the attacker sells the asset at a higher price with the second transaction they placed, profiting from the artificially created price differential. This type of exploit takes advantage of the transparency and immutability of blockchains, where pending transactions can be seen by all but are irrevocable once initiated.

```
SW_Attack
let SW(o,sdr,pair,amt,amt'):= Order(o) ∧
    Sender(s) ∧ Swap(pair,amt,amt') in
    SW(o₂,sdr₂,pair₂,amt₂,amt'₂) ∧
    ♦_I SW(o₂,sdr₂,pair₂,amt₂,amt'₂) ∧
    ◇_I SW(o₂,sdr₂,pair₂,amt₂,amt'₂)
    s₁ = s₃ ∧ ¬ (s₁ = s₂) ∧
    pair₁ = pair₃ ∧ ¬ (pair₁ = pair₂) ∧
    amt₃ = amt'₁ ∧ amt'₃ > amt₁
```

A Sandwich attack is a type of multi-transaction attack commonly captured in exchanges with frequent token swaps, typically Uniswap. In the MFOTL formula, the victim s_2 executes a token swap for a specific token pair $pair_2$, which is described by the SW predicate in the formula. A malicious attacker s_1 detects this pending transaction and, beforehand, initiates a similar swap for the same token pair $pair_1$ ($pair_1$=$pair_2$), acquiring a specific asset and inflating its price. This causes the victim s_2 to receive fewer assets. Subsequently, the attacker initiates a second transaction, executing a reverse swap ($pair_3 \neq pair_1$), selling the previously acquired assets and profiting from the created price discrepancy.

E Derivation for Quantitative Semantics

In the following, we show that how we can derive Definition 1 in the paper and how we relate it to the common (bool) satisfaction relation. We include some intuitions to ease comprehension for readers.

The subset MFOTL syntax used in this paper is: Here event $e_B(\bar{t})$

$$\varphi_{atm} ::= e_A(\bar{t}) \mid e_B(\bar{t}) \mid \mathrm{op}(t,t)$$
$$\mathrm{op}(t,t) ::= t = t \mid t < t,\ t \in \mathbf{Dom}(e_A) \cup \mathbf{Dom}(e_B)$$
$$\varphi ::= \varphi_{atm} \mid \varphi \wedge \varphi \mid \Diamond_I \varphi_{atm} \mid \blacklozenge_I \varphi_{atm} \mid \neg \varphi_{atm}.$$

Figure 8: The subset MFOTL syntax used in this paper.

denotes the *Basic Transaction Semantics (BTS)* and $e_A(\bar{t})$ denotes the *Advanced Attack Semantics (AAS)*. For simplicity, we use $t \in \mathbf{Dom}(e)$ to indicate that t is an argument of event e.

In this paper, every attack formula φ can be unfolded using the conjunction operator as follows:

$$\varphi := \varphi_1 \ \wedge\ \varphi_2 \ \wedge\ \ldots \wedge\ \varphi_n,$$

where each φ_i is either φ_{atm}, $\blacklozenge_I \varphi_{atm}$, $\mathrm{op}(t,t)$, or $\Diamond_I \varphi_{atm}$.

Our intuition is based on the idea that the satisfaction of each subformula φ_i contributes to the satisfaction of the top-level formula φ. This is similar to *Łukasiewicz fuzzy logic* [1], which differs from

[1] In Łukasiewicz fuzzy logic, conjunction is handled by taking the maximum of 0 and the sum of the two truth values minus 1, in contrast to classical fuzzy logic (Zadeh fuzzy logic), which uses the minimum of the two truth values for conjunction.

classical fuzzy logic, as mentioned in our email. Thus, the quantitative satisfaction value of φ equals the sum of the satisfaction values of all its sub-formulas, i.e.,

$$\varrho(\varphi, i, \Pi) = \sum_j \varrho(\varphi_j, i, \Pi) = \varrho(\varphi_1, i, \Pi) + \varrho(\varphi_2, i, \Pi) + \cdots + \varrho(\varphi_n, i, \Pi) \quad \textbf{(I1)}$$

Now we attempt to relate this to the standard (Boolean) satisfaction relation. Given a semantic log Π, a time index i, and an attack formula φ, we have two main intuitions:

- *intuition 1*: If $\exists v$ such that $v, i \models_\Pi \varphi$ (i.e., φ can be satisfied at index i), then the value of the quantitative satisfaction of φ should be 1;
- *intuition 2*: If none of the sub-formulas φ_i can be satisfied, then the value of the quantitative satisfaction of φ should be 0. (This is somewhat akin to the log being *independent* to φ at index i).

The term 'independent' used in the second intuition can be interpreted as 'complete unsatisfiability.'

Now, our task is as follows: Given a formula φ, if we can assign a quantified value to each atomic sub-formula φ_i when it is satisfied (i.e., its 'contribution' to the overall satisfaction of the formula), we can then quantify the degree of satisfaction of the entire formula accordingly.

We use c_i ($c_i > 0$) to denote the 'contribution' of each formula unit φ_i when it is satisfied, i.e.,

$$\varrho(\varphi_i, j, \Pi) = c_i, \ \textbf{if} \ \exists v.\ v, j \models_\Pi \varphi_i. \quad \textbf{(I1)}$$

If it cannot be satisfied, it makes no contribution to the overall formula, thus,

$$\varrho(\varphi_i, j, \Pi) = 0, \ \textbf{if} \ \forall v.\ v, j \not\models_\Pi \varphi_i. \quad \textbf{(I2)}$$

When φ is satisfied for a given valuation v at index i, i.e.,

$$v, i \models_\Pi \varphi,$$

and since $\forall i,\ \varphi \Rightarrow \varphi_i$, we can infer that v is also a valuation for all φ_i, i.e.,

$$\forall i,\ v, i \models_\Pi \varphi_i.$$

According to *intuition* E, we have the following relation:

$$\sum_i \varrho(\varphi_i, j, \Pi) = \sum_i c_i = 1.$$

From this relation, we know that our task is to *distribute* the total satisfaction value of 1 across the formula units φ_i, i.e., to determine each constant c_i for a given formula φ. This distribution process is similar to a *normalization process*, where we ensure that the following condition is always satisfied:

$$\sum_i c_i = 1. \quad (*)$$

Distribution Approach. Recall the definition of φ:

$$\varphi := \varphi_1 \ \wedge \ \varphi_2 \ \wedge \ \ldots \wedge \ \varphi_n,$$

where each φ_i is either φ_{atm}, $\blacklozenge_I \varphi_{\text{atm}}$, or $\Diamond_I \varphi_{\text{atm}}$. We classify φ into two distinct types:

(1) There is only one kind of semantic event in φ_{a}, either *BTS* or *AAS*.
(2) Both semantic events, *BTS* and *AAS*, occur simultaneously in φ.

In the first case, we evenly distribute the total value of 1 across each sub-formula, i.e., $c_i = 1/n$ (where n is the number of atomic formulas). The ***intuition*** behind this is that the satisfaction of each sub-formula contributes equally to the overall satisfaction.

In the second case, we first determine the contribution value of each sub-formula when it is satisfied, and then distribute the total value of 1 among them based on these contributions.

In this work, the attack formula φ always falls into the second case, i.e., both semantic events, *BTS* and *AAS*, occur simultaneously in φ_{a}. Now, we introduce ω_a and ω_b along with the *distribution algorithm*.

Algorithm for Case 2. We define ω_a and ω_b ($\omega_a > \omega_b$) as the sum of the contribution values of all sub-formulas related to *AAS* and *BTS*, respectively where:

$$\omega_a + \omega_b = 1,$$
$$\sum_{i \in \mathcal{I}_a} c_i = \omega_a,$$
$$\sum_{i \in \mathcal{I}_b} c_i = \omega_b$$

The corresponding *intuition* for $\omega_a > \omega_b$ is that: *the satisfaction of AAS-related atomic formulas makes a larger contribution than BTS-related atomic formulas to the overall satisfaction.*

We use $\mathcal{I}_a$ and $\mathcal{I}_b$ to denote the index sets of *AAS-related atomic formulas* and *BTS-related atomic formulas* within attack formula φ, respectively. Specifically, we define:

$$\mathcal{I}_b = \{i \mid \varphi_i := e_{\text{B}}(\bar{t})\} \cup \quad \textbf{(D1)}$$
$$\{i \mid \varphi_i := \Diamond_I e_{\text{B}}(\bar{t})\} \ \cup \ \{i \mid \varphi_i := \blacklozenge_I e_{\text{B}}(\bar{t})\} \cup \quad \textbf{(D2)}$$
$$\{i \mid \varphi_i := \text{op}(t_1, t_2), \text{ and } t_1, t_2 \in \textbf{Dom}(e_{\text{B}})\}. \quad \textbf{(D3)}$$

In the above definition, **D3** specifies that φ_i is a formula $\text{op}(t_1, t_2)$, where t_1 and t_2 are arguments of BTS events (i.e., $t_1, t_2 \in \textbf{Dom}(e_{\text{B}})$).

Similarly, we define:

$$\mathcal{I}_a = \{i \mid \varphi_i := e_{\text{A}}(\bar{t})\} \cup$$
$$\{i \mid \varphi_i := \Diamond_I e_{\text{A}}(\bar{t})\} \ \cup \ \{i \mid \varphi_i := \blacklozenge_I e_{\text{A}}(\bar{t})\} \cup$$
$$\{i \mid \varphi_i := \text{op}(t_1, t_2), \text{ and } t_1, t_2 \in \textbf{Dom}(e_{\text{A}})\}.$$

As in the first case, we evenly distribute the total values of ω_a and ω_b across the sub-formulas, i.e.,

$$c_i = \begin{cases} \frac{\omega_a}{|\mathcal{I}_a|}, & \textbf{if} \ i \in \mathcal{I}_a \\ \frac{\omega_b}{|\mathcal{I}_b|}, & \textbf{if} \ i \in \mathcal{I}_b. \end{cases} \quad \textbf{(C1)}$$

Clearly, this satisfies the following condition:

$$\sum_i c_i = \sum_{i \in \mathcal{I}_a} c_i + \sum_{i \in \mathcal{I}_b} c_i = \frac{\omega_a}{|\mathcal{I}_a|} \cdot |\mathcal{I}_a| + \frac{\omega_b}{|\mathcal{I}_b|} \cdot |\mathcal{I}_b| = 1.$$

Combine the **I0**, **I1** and **I2**, we have:

$$\varrho(\varphi_1 \wedge \varphi_2, i, \Pi) = \varrho(\varphi_1, i, \Pi) + \varrho(\varphi_2, i, \Pi), \quad \textbf{(R1)}$$
$$\varrho(\varphi_{\text{atm}}, i, \Pi) = \omega_a / |I_a|, \ \textbf{if} \ \exists v.\ v, \ i \models_\Pi \varphi_{\text{atm}} \ \textbf{and} \ \varphi_{\text{atm}} \in \Phi_a \quad \textbf{(R2)}$$
$$\varrho(\varphi_{\text{atm}}, i, \Pi) = \omega_b / |I_b|, \ \textbf{if} \ \exists v.\ v, \ i \models_\Pi \varphi_{\text{atm}} \ \textbf{and} \ \varphi_{\text{atm}} \in \Phi_b \quad \textbf{(R3)}$$
$$\varrho(\varphi_{\text{atm}}, i, \Pi) = 0, \ \textbf{if} \ \forall v.\ v, \ i \not\models_\Pi \varphi_{\text{atm}}. \quad \textbf{(R4)}$$

where Φ_a and Φ_b denotes the two different types of formulas (AAS-related formulas and BTS-related formulas) respectively, i.e.,

$$\Phi_a = \{\varphi := e_{\text{A}}(\bar{t})\} \cup$$
$$\{\varphi := \Diamond_I e_{\text{A}}(\bar{t})\} \ \cup \ \{\varphi := \blacklozenge_I e_{\text{A}}(\bar{t})\} \cup$$
$$\{\varphi := \text{op}(t_1, t_2), \text{ and } t_1, t_2 \in \textbf{Dom}(e_{\text{A}})\}.$$

and

$$\begin{aligned}\Phi_b = &\{\varphi := e_{\mathrm{B}}(\bar{t})\} \cup \\ &\{\varphi := \Diamond_I e_{\mathrm{B}}(\bar{t})\} \cup \{\varphi := \blacklozenge_I e_{\mathrm{B}}(\bar{t})\} \cup \\ &\{\varphi := \mathrm{op}(t_1, t_2), \text{ and } t_1, t_2 \in \mathbf{Dom}(e_{\mathrm{B}})\}.\end{aligned}$$

The definition **R1-R4** is close to Definition 1 in our paper. Now we further handle the temporal operator. We take $\Diamond_I e_{\mathrm{A}}$ as an example to show how we can further quantities it. According to the definition above, we have

$$\varrho(\Diamond_I e_{\mathrm{A}}, i, \Pi) = \omega_a / |I_a|, \text{ if } \exists v.\ v,\ i \models_\Pi \Diamond_I e_{\mathrm{A}}.$$

But consider the normal satisfaction relation:

$$v, i \models_\Pi \Diamond_I \varphi, \text{ if } v, j \models_\Pi \varphi \text{ for some } j \geq i,\ \tau_j - \tau_i \in I.$$

Our intuition for the quantitative semantics of $\Diamond_I \varphi_{\mathrm{atm}}$ is that the *faster* φ_{atm} is satisfied within the interval I, the greater its contribution to the satisfaction of the overall formula φ.

Real-World Rationale. This is because the state of the Ethereum blockchain is constantly evolving in real time, and network participants rely on the current state of resources when executing transactions. If attack behaviors are prolonged over time, the network's resource state, contract conditions, or token prices may fluctuate, potentially affecting the effectiveness and success rate of the attack. Therefore, the faster φ_{atm} is satisfied, the more significant its contribution to determining whether a transaction constitutes an attack.

Thus, we propose a more refined quantitative approach to capture this intuition:

$$\varrho(\Diamond_I \varphi_{\mathrm{atm}}, i, \Pi) = \max_{j \text{ s.t. } \tau_j - \tau_i < I} \left\{ \varrho(\varphi_{\mathrm{atm}}, j, \Pi) \cdot \frac{|I \vdash (\tau_j - \tau_i)}{|I|} \right\}. \tag{R5}$$

Similarly, for the temporal operator $\blacklozenge_I$, we have:

$$\varrho(\blacklozenge_I \varphi_{\mathrm{atm}}, i, \Pi) = \max_{j \text{ s.t. } \tau_i - \tau_j < I} \left\{ \varrho(\varphi_{\mathrm{atm}}, j, \Pi) \cdot \frac{|I \vdash (\tau_i - \tau_j)}{|I|} \right\}. \tag{R6}$$

Here we use $\frac{|I \vdash (\tau_i - \tau_j)}{|I|}$ to measure how *faster* the formula φ_{atm} can be satisfied. Now we get the final definition of the function ϱ, i.e., the **Definition 1** in our paper.

Definition 1. *Given an MFOTL formula φ and an event set π_i within a semantic log Π, the quantitative semantics can be evaluated using a function ϱ, which is defined recursively as follows:*

$$\varrho(\varphi_{\mathrm{atm}}, i, \Pi) = \omega_a / |\mathcal{I}_a|, \text{ if } \exists v.\ v,\ i \models_\Pi \varphi_{\mathrm{atm}} \text{ and } \varphi_{\mathrm{atm}} \in \Phi'_a, \tag{T1}$$

$$\varrho(\varphi_{\mathrm{atm}}, i, \Pi) = \omega_b / |\mathcal{I}_b|, \text{ if } \exists v.\ v,\ i \models_\Pi \varphi_{\mathrm{atm}} \text{ and } \varphi_{\mathrm{atm}} \in \Phi'_b, \tag{T2}$$

$$\varrho(\varphi_{\mathrm{atm}}, i, \Pi) = 0, \text{ if } \forall v.\ v,\ i \not\models_\Pi \varphi_{\mathrm{atm}}. \tag{T3}$$

$$\varrho(\Diamond_I \varphi_{\mathrm{atm}}, i, \Pi) = \max_{j \text{ s.t. } \tau_j - \tau_i < I} \left\{ \varrho(\varphi_{\mathrm{atm}}, j, \Pi) \cdot \frac{|I| - (\tau_j - \tau_i)}{|I|} \right\}, \tag{T4}$$

$$\varrho(\blacklozenge_I \varphi_{\mathrm{atm}}, i, \Pi) = \max_{j \text{ s.t. } \tau_i - \tau_j < I} \left\{ \varrho(\varphi_{\mathrm{atm}}, j, \Pi) \cdot \frac{|I| - (\tau_i - \tau_j)}{|I|} \right\}, \tag{T5}$$

$$\varrho(\varphi_1 \wedge \varphi_2, i, \Pi) = \varrho(\varphi_1, i, \Pi) + \varrho(\varphi_2, i, \Pi). \tag{T6}$$

Here, the relation $v, i \models_\Pi \varphi$ denotes the satisfaction of the formula φ for a valuation v at an index i with respect to the trace Π. We have:

$$\Phi'_a = \{\varphi \mid \varphi := e_{\mathrm{A}}(\bar{t})\} \cup \{\varphi \mid \varphi := \mathrm{op}(t_1, t_2), \text{ and } t_1, t_2 \in \mathbf{Dom}(e_{\mathrm{A}})\},$$

and

$$\Phi'_b = \{\varphi \mid \varphi := e_{\mathrm{B}}(\bar{t})\} \cup \{\varphi \mid \varphi := \mathrm{op}(t_1, t_2), \text{ and } t_1, t_2 \in \mathbf{Dom}(e_{\mathrm{B}})\}.$$

In fact the above definition is adequate to cover all the cases, to make it complete, we can add a definition for $\neg$ operator:

$$\varrho(\neg\varphi_{\mathrm{atm}}, i, \Pi) = \omega_b / |\mathcal{I}_b| - \varrho(\varphi_{\mathrm{atm}}, i, \Pi), \tag{T7}$$

$$\varrho(\neg\varphi_{\mathrm{atm}}, i, \Pi) = \omega_a / |\mathcal{I}_a| - \varrho(\varphi_{\mathrm{atm}}, i, \Pi). \tag{T8}$$

The definition of ϱ in **R1-R4** is extended to **T1-R8** (the definition with $\Diamond$ and $\blacklozenge$ operators has changed). Now we show that the intuitions introduced before still holds and the computation result of ϱ still lie in [0,1].

Properties of ϱ. *According to Definition 1, given an attack formula φ and a semantic log Π, the function ϱ always returns a value within the range [0, 1] at any index i, i.e., $\varrho(\varphi, i, \Pi) \in [0, 1]$.*

Proof. Unfold ϱ thoroughly with **T8** in Definition 1, we have:

$$\varrho(\varphi, i, \Pi) = \varrho(\varphi_1, i, \Pi) + \varrho(\varphi_2, i, \Pi) + \ldots + \varrho(\varphi_n, i, \Pi)$$

where φ_i is atomic formula φ_{atm}. Similar to **D1-D3**, we introduce $\mathcal{I}'_a$ and $\mathcal{I}'_\downarrow$ to denote the indexes sets of *AAS-related* atomic formulas and BTS-related atomic formulas without temporal operators, i.e,

$$\begin{aligned}\mathcal{I}_b &= \mathcal{I}'_b \cup \{i \mid \varphi_i := \Diamond_I e_{\mathrm{B}}(\bar{t})\} \cup \{i \mid \varphi_i := \blacklozenge_I e_{\mathrm{B}}(\bar{t})\} \\ \mathcal{I}_a &= \mathcal{I}'_a \cup \{i \mid \varphi_i := \Diamond_I e_{\mathrm{A}}(\bar{t})\} \cup \{i \mid \varphi_i := \blacklozenge_I e_{\mathrm{A}}(\bar{t})\}.\end{aligned}$$

For further simplicity, we use notation:

$$\begin{aligned}\mathcal{I}_{\Diamond,b} &= \{i \mid \varphi_i := \Diamond_I e_{\mathrm{B}}(\bar{t})\}, \quad \mathcal{I}_{\blacklozenge,b} = \{i \mid \varphi_i := \blacklozenge e_{\mathrm{B}}(\bar{t})\} \\ \mathcal{I}_{\Diamond,a} &= \{i \mid \varphi_i := \Diamond_I e_{\mathrm{A}}(\bar{t})\}, \quad \mathcal{I}_{\blacklozenge,a} = \{i \mid \varphi_i := \blacklozenge e_{\mathrm{A}}(\bar{t})\}\end{aligned}$$

We have that:

$$\begin{aligned}|\mathcal{I}_b| &= |\mathcal{I}'_b| + |\mathcal{I}_{\Diamond,b}| + |\mathcal{I}_{\blacklozenge,b}| \\ |\mathcal{I}_a| &= |\mathcal{I}'_a| + |\mathcal{I}_{\Diamond,a}| + |\mathcal{I}_{\blacklozenge,a}|\end{aligned}$$

We first prove the maximum of $\varrho(\varphi, i, \Pi)$ is 1. Assume that for any $i \in \mathcal{I}'_a \cup \mathcal{I}'_b$, the corresponding formula is satisfied (i.e., $\forall i \in \mathcal{I}'_a \cup \mathcal{I}'_b.\ \exists v.\ v, i \models_\Pi \varphi_i$), according to **T1** and **T2**, we have:

$$\begin{aligned}&\varrho(\varphi, i, \Pi) \\ &= \sum_{i \in \mathcal{I}'_a} c_i + \sum_{i \in \mathcal{I}_a / \mathcal{I}'_a} \varrho(\varphi_i, i, \Pi) + \sum_{i \in \mathcal{I}'_b} c_i + \sum_{i \in \mathcal{I}_a / \mathcal{I}'_b} \varrho(\varphi_i, i, \Pi) \\ &= \frac{\omega_a}{|\mathcal{I}_a|} \cdot |\mathcal{I}'_a| + \sum_{i \in \mathcal{I}_a / \mathcal{I}'_a} \varrho(\varphi_i, i, \Pi) + \frac{\omega_b}{|\mathcal{I}_b|} \cdot |\mathcal{I}'_b| + \sum_{i \in \mathcal{I}'_a / \mathcal{I}'_b} \varrho(\varphi_i, i, \Pi) \\ &= \frac{\omega_a}{|\mathcal{I}_a|} \cdot |\mathcal{I}'_a| + \sum_{i \in \mathcal{I}_{\Diamond,a}} \varrho(\varphi_i, i, \Pi) + \sum_{i \in \mathcal{I}_{\blacklozenge,a}} \varrho(\varphi_i, i, \Pi) \\ &+ \frac{\omega_b}{|\mathcal{I}_b|} \cdot |\mathcal{I}'_b| + \sum_{i \in \mathcal{I}_{\Diamond,b}} \varrho(\varphi_i, i, \Pi) + \sum_{i \in \mathcal{I}_{\blacklozenge,b}} \varrho(\varphi_i, i, \Pi)\end{aligned}$$

According to **T4** and **T5**, for any $i \in \mathcal{I}_{\Diamond,a}$ (where $\varphi_i := \Diamond e_{\mathrm{A}}(\bar{t})$), we have:

$$\varrho(\Diamond_I e_{\mathrm{A}}(\bar{t}), i, \Pi) = \max_{j \text{ s.t. } \tau_j - \tau_i < I} \left\{ \varrho(e_{\mathrm{A}}(\bar{t}), j, \Pi) \cdot \frac{|I| - (\tau_j - \tau_i)}{|I|} \right\} \leq \frac{|\omega_a|}{|\mathcal{I}_a|},$$

$$\varrho(\blacklozenge_I e_{\mathrm{A}}(\bar{t}), i, \Pi) = \max_{j \text{ s.t. } \tau_i - \tau_j < I} \left\{ \varrho(e_{\mathrm{A}}(\bar{t}), j, \Pi) \cdot \frac{|I| - (\tau_i - \tau_j)}{|I|} \right\} \leq \frac{|\omega_a|}{|\mathcal{I}_a|}$$

Similarly, we have:

$$\varrho(\Diamond_I e_{\mathrm{B}}(\bar{t}), i, \Pi) = \max_{j \text{ s.t. } \tau_j - \tau_i < I} \left\{ \varrho(e_{\mathrm{B}}(\bar{t}), j, \Pi) \cdot \frac{|I| - (\tau_j - \tau_i)}{|I|} \right\} \leq \frac{|\omega_b|}{|\mathcal{I}_b|},$$

$$\varrho(\blacklozenge_I e_{\mathrm{B}}(\bar{t}), i, \Pi) = \max_{j \text{ s.t. } \tau_i - \tau_j < I} \left\{ \varrho(e_{\mathrm{A}}(\bar{t}), j, \Pi) \cdot \frac{|I| - (\tau_i - \tau_j)}{|I|} \right\} \leq \frac{|\omega_b|}{|\mathcal{I}_b|}.$$

For the sub-formula $\Diamond_I \varphi_{\text{atm}}$, consider φ_{atm} is always true, then we obtain:

$$\varrho(\blacklozenge_I \varphi_{\text{atm}}, i, \Pi) = \varrho(\varphi_{\text{atm}}, i, \Pi) = \begin{cases} \frac{|\omega_b|}{|\mathcal{I}_b|}, & \textbf{if } \varphi_{\text{atm}} \in \Phi'_b \\ \frac{|\omega_a|}{|\mathcal{I}_a|}, & \textbf{if } \varphi_{\text{atm}} \in \Phi'_a \end{cases}$$

Combine above relations, we have:

$$\begin{aligned} \varrho(\varphi, i, \Pi) \leq & \frac{\omega_b}{|\mathcal{I}_b|} \cdot |\mathcal{I}'_b| + \frac{\omega_b}{|\mathcal{I}_b|} \cdot |\mathcal{I}_{\Diamond,b}| + \frac{\omega_b}{|\mathcal{I}_b|} \cdot |\mathcal{I}_{\blacklozenge,b}| \\ & + \frac{\omega_a}{|\mathcal{I}_a|} \cdot |\mathcal{I}'_a| + \frac{\omega_a}{|\mathcal{I}_a|} \cdot |\mathcal{I}_{\Diamond,a}| + \frac{\omega_a}{|\mathcal{I}_a|} \cdot |\mathcal{I}_{\blacklozenge,a}| \\ & = 1. \end{aligned}$$

According to the Definition 1, $\varrho(\varphi, i, \Pi)$ cannot be a negative time, i.e., $\varrho(\varphi, i, \Pi) \geq 0$.

Thus, we prove that the property illustrated above for ϱ is satisfied.

F Illustrative Examples

Given a MFOTL formula φ:

$$\varphi := e_1(n_1) \ \wedge \ e_2(n_2) \ \wedge \Diamond_{[0,3)} e_3(n_3) \ \wedge \ \ n_1 > n_3$$

We define

$$\begin{aligned} \varphi_1 &:= e_1(n_1), \ \textit{AAS-related atomic formula} \\ \varphi_2 &:= e_2(n_2), \ \textit{BTS-related atomic formula} \\ \varphi_3 &:= e_3(n_3), \ \textit{AAS-related} \\ \varphi_4 &:= n_1 > n_3, \ (n_1, n_3 \in \textbf{Dom}(e_{\text{A}})) \end{aligned}$$

We get $\mathcal{I}_a = \{1, 3, 4\}$ and $\mathcal{I}_b = \{2\}$. Given an event set π_i within semantic log $\Pi = \langle \pi_1, ...\pi_4 \rangle$:

$$\begin{aligned} \pi_1 &:= @1 \quad e_2(1) \\ \pi_2 &:= @2 \quad e_1(1) \\ \pi_3 &:= @3 \quad e_1(3) \quad e_2(1) \\ \pi_4 &:= @4 \quad e_3(1) \end{aligned}$$

We set $w_a = 0.9$, $w_b = 0.1$, according to **C1**, we get:

$$\begin{aligned} c_1 = c_3 = c_4 &= \omega_a / |I_a| = 0.9/3 = 0.3 \\ c_2 &= \omega_b / |I_b| = 0.1/1 = 0.1 \end{aligned}$$

To calculate the quantitative value $\varrho(\varphi, i, \Pi)$, the process is as follows.

$$\varrho(\varphi, i, \Pi) = \varrho(\varphi_1, i, \Pi) + \varrho(\varphi_2, i, \Pi) + \varrho(\Diamond_{[0,3)} \varphi_3, i, \Pi) + \varrho(\varphi_4, i, \Pi)) \qquad \textbf{(T8)}$$

$$\varrho(\varphi_1, i, \Pi) = \begin{cases} c_1, & \text{if } \exists v.\ v, i \models_\Pi \varphi_1 \\ 0, & \text{if } \forall v.\ v, i \not\models_\Pi \varphi_1 \end{cases} \qquad \textbf{(T1,T3)}$$

$$\varrho(\varphi_2, i, \Pi) = \begin{cases} c_2, & \text{if } \exists v.\ v, i \models_\Pi \varphi_2 \\ 0, & \text{if } \forall v.\ v, i \not\models_\Pi \varphi_2 \end{cases} \qquad \textbf{(T2,T3)}$$

$$\varrho(\Diamond_{I=[0,3)} \varphi_3, i, \Pi) = \max_{j \text{ s.t. } \tau_j - \tau_i < I} \{\varrho(\varphi_3, j, \Pi) \cdot \frac{|I| - (\tau_j - \tau_i)}{|I|}\},$$

$$= \max\left(\varrho(\varphi_3, i, \Pi) \cdot \frac{3-0}{3}, \varrho(\varphi_3, \text{i}+1, \Pi) \cdot \frac{3-1}{3}, \varrho(\varphi_3, \text{i}+2, \Pi) \cdot \frac{3-2}{3}\right)$$

$$\textbf{if } \exists v.v, j \models_\Pi \varphi_3, \qquad \textbf{(T6)}$$

$$\varrho(\varphi_3, i, \Pi) = \begin{cases} c_3, & \text{if } \exists v.\ v, i \models_\Pi \varphi_3 \\ 0, & \text{if } \forall v.\ v, i \not\models_\Pi \varphi_3 \end{cases} \qquad \textbf{(T1,T3)}$$

$$\varrho(\varphi_4, i, \Pi) = \begin{cases} c_4, & \text{if } \exists v.\ v, i \models_\Pi \varphi_4 \\ 0, & \text{if } \forall v.\ v, i \not\models_\Pi \varphi_4 \end{cases} \qquad \textbf{(T1,T3)}$$

The results are as follows:

$$\begin{aligned} \varrho(\varphi, 1, \Pi) &= 0.1 \\ \varrho(\varphi, 2, \Pi) &= 0.3 + \frac{3-2}{3} \cdot 0.3 = 0.4 \\ \varrho(\varphi, 3, \Pi) &= 0.3 + 0.1 + \frac{3-1}{3} \cdot 0.3 + 0.3 = 0.9 \\ \varrho(\varphi, 4, \Pi) &= 0.3 \end{aligned}$$

G Experiments

Table 9: Time Consumption

Attack Type	Tracing	Lifting	Monitoring	Total
Sandwich			1.12ms	6.45ms
Reentrancy			11.57ms	16.90ms
CI	5.20ms	0.13ms	15.21ms	20.54ms
DPM			18.10ms	23.43ms
IPM			15.12ms	20.45ms