Evil Under the Sun: Understanding and Discovering Attacks on Ethereum Decentralized Applications

Liya Su, *Indiana University Bloomington; Institute of Information Engineering, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Xinyue Shen, *Indiana University Bloomington and Alibaba Group;* Xiangyu Du, *Indiana University Bloomington; Institute of Information Engineering, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Xiaojing Liao, XiaoFeng Wang, and Luyi Xing, *Indiana University Bloomington;* Baoxu Liu, *Institute of Information Engineering, Chinese Academy of Sciences; University of Chinese Academy of Sciences*

https://www.usenix.org/conference/usenixsecurity21/presentation/su

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

Evil Under the Sun: Understanding and Discovering Attacks on Ethereum Decentralized Applications

Liya Su[1,2,3] *†, Xinyue Shen[1,4]*†, Xiangyu Du[1,2,3]*, Xiaojing Liao[1],
XiaoFeng Wang[1], Luyi Xing[1], Baoxu Liu[2,3]
[1]*Indiana University Bloomington,* [2]*Institute of Information Engineering, Chinese Academy of Sciences,*
[3]*University of Chinese Academy of Sciences,* [4]*Alibaba Group*
{liyasu, shen12, duxian}@iu.edu, {xliao, xw7, luyixing}@indiana.edu, liubaoxu@iie.ac.cn

Abstract

The popularity of Ethereum decentralized applications (Dapps) also brings in new security risks: it has been reported that these Dapps have been under various kinds of attacks from cybercriminals to gain profit. To the best of our knowledge, little has been done so far to understand this new cybercrime, in terms of its scope, criminal footprints and attack operational intents, not to mention any efforts to investigate these attack incidents automatically on a large scale. In this paper, we performed the first measurement study on real-world Dapp attack instances to recover critical threat intelligence (e.g., kill chain and attack patterns). Utilizing such threat intelligence, we proposed the first technique *DEFIER* to automatically investigate attack incidents on a large scale. Running *DEFIER* on 2.3 million transactions from 104 Ethereum on-chain Dapps, we were able to identify 476,342 exploit transactions on 85 target Dapps, which related to 75 0-day victim Dapps and 17K previously-unknown attacker EOAs. To the best of our knowledge, it is the largest Ethereum on-chain Dapp attack incidents dataset ever reported.

1 Introduction

The rise of blockchain technologies has profoundly transformed computing, bringing to the front a new type of *decentralized applications* on blockchain that facilitate transfer of values across users without a third party. Such applications, dubbed *Dapp*, have already been widely deployed on *Ethereum* to provide services ranging from cryptocurrency management to voting and governance [17]. Online statistics show that till Nov. 5, 2019, 3,137 Dapps on Ethereum are serving 63.77K active users every day through over one million transactions that involve 7.55 million USD [2]. However, the boundless potentials Dapps have opened also come with new security risks. It has been reported that cybercriminals have fixed their gaze on Dapps and exploits on them, particularly their blockchain back-end (i.e., *smart contracts*, see Section 2), happening from time to time. A prominent example is the DAO attack that caused a loss over 50 million USD [39] in 2016, resulting in the hard-fork in Ethereum. Also found in our study is that miscreants took 14K Ethers from the victim Dapps with most financial losses (i.e., Fomo3D, Section 4.5).

With this significant threat, the community's understanding about the new type of cybercrimes is still very limited: to the best of our knowledge, no extensive forensic analysis on Dapp attacks has ever been reported, nor has any *cyber threat intelligence* (CTI) been collected from them to find out the perpetrator's strategy, capability and infrastructure, not to mention to utilize the knowledge to mitigate the threat.

Understanding attacks on Dapps. In this paper, we present the first study that analyzes and measures real-world attacks on Ethereum Dapps based upon the forensic evidence recorded on the blockchain, which brings new insights to this emerging cybercrime. Our research leverages the information logged by the Ethereum blockchain, an open, immutable ledger recording the entire history of interactions between Dapps and their users through their Ethereum user accounts (i.e., Externally Owned Accounts or EOA, see Section 2). Such interactions are performed through *transactions*, which are logged in the data packages chained by Ethereum. Should a Dapp be exploited, all forensic evidence, such as attack traces, will be kept in related transactions, which can later be used to analyze the attack.

However, it is nontrivial to identify attack traces from over 350 million Ethereum transactions. Finding related transactions from published reports is inadequate at best, since they tend to miss information about important actors and exploit behaviors (such as exploit developers, Section 4), when their EOAs are not included in the reports. Also absent are detailed internal operations triggered by each transaction, in terms of function calls between the target Dapps and EOAs or between different EOAs (see Section 4). Such calls describe these parties' behaviors and are found to be critical for determining their intents during the interactions. To address these challenges, we come up with a methodology that utilizes known attack-related transactions (called *exploit transactions* in the paper) and EOAs to find new ones and further analyze their execution traces (by re-executing these transactions). In this way, utilizing 25 Dapps related to 42 known attack incidents, we identified 58,555 exploit transactions with 436,371 execution

*This work was done when the student authors were in Indiana University Bloomington

†First two authors contributed equally for this project

traces, all linked to 56 Dapps, including 29 being exploited but *never reported before* (called *0-day* victim Dapps).

Cyber threar intelligence

Our findings. From the transactions collected, our forensic analysis has recovered critical CTI about strategically, well organized Dapp attacks, which have *never* been done before. Such threat information (CTI) provides invaluable insights for understanding the strategies, approaches and intentions of real-world cybercriminals in attacking Dapps, and thus contributes to mitigating the emerging threats. Most interesting is the discovery about how the adversary systematically orchestrates an attack. More specifically, across different kinds of exploits (weak randomness exploit, denial of services, integer overflow, reentrancy and authentication circumvention) against different Dapps, we can see a general attack lifecycle with four stages from the transaction sequences involved: attack preparation, exploitation, propagation and completion. These stages form a *kill chain* against Dapps, which has never been reported before. The chain starts with repeated attempts to probe the target Dapp from various sources for finding and testing its vulnerable functions. That is, the adversary tests, debugs the attack code to ensure it can successfully exploit the particular target Dapp. This stage is followed by a series of exploit transactions to profit from the target, which are continuously refined to improve efficiency. After that, the same attack is often replayed to similar Dapps, with a sequence of transactions produced to aim at different targets. The attack is finalized with another sequence of transactions for terminating attack contracts and transferring stolen funds. Across different attack instances against real-world Dapps, this lifecycle paradigm exhibits remarkable consistency, with each stage characterized by a time series of similar, inter-dependent transactions executed consecutively within a short time window. The series describes the adversary's behaviors and thus characterizes his intent at each stage. For example, continuous probing transactions show the intent of finding weaknesses in a target Dapp.

kill chain descibe

Further, our research reveals a hierarchical attack infrastructure with multiple roles working together to execute different types of exploits. These roles include exploit developer (testing an attack on vulnerable functions/Dapps), attack operator (executing an exploit through attack transactions), money mule (helping profit/attack cost transfers through an anonymity channel [18]) and money manager (managing profit/cost transfers). Each of them has well-defined tasks and therefore behaves similarly across different attack types and instances. This again makes their execution traces exhibit some level of homogeneity at each attack stage.

Extended attack discovery and investigation. The CTI (e.g., kill chain and operational intents) recovered in our study can potentially lead to the exposure of unknown threats to Dapps. To understand the values of our findings, we designed an *exploit discovery* methodology, called *DEFIER* (Dapp Exploit Investigator), to find more attack instances, particularly those never reported, so as to gain more insights into real-world attacks on Dapps. *DEFIER* captures the adversary's strategies and intents, as demonstrated by the operations triggered by the transaction time series at each stage. Given a Dapp, our approach first gathers all its transactions recorded on the blockchain and from them, further finds out other related transactions and EOAs. All these transactions are then clustered based upon the similarity of their execution traces in a graph form and organized into several time series. After converting the execution traces of each transaction into a vector through graph embedding, we run a Long Short-Term Memory (LSTM) neural network to classify each time series, which determines not only whether the series is related to an exploit, but also its attack stage when it is.

Running *DEFIER* on 104 Dapps, we were able to discover 476,342 exploit transactions on 85 target (with a micro-precision of 91.7%). In particular, *DEFIER* reported 75 *0-day victim Dapps* (e.g., SpaceWar and SuperCard). Also surprisingly, our study shows that a substantial portion (i.e., 26%) of the transactions of these Dapps (on Ethereum) are attack-related: e.g., 30% of Fomo3D's transactions are attack-related (from July 2018 to April 2019). This provides evidence that indeed the attack lifecycle we discovered is general. Such an attack lifecycle discovery tool can potentially be used to disrupt exploits, sometimes even before damages are inflicted (e.g., finding and stopping an attack at its preparation stage).

Contribution. The contributions of the paper are as follows:

- We performed the *first* measurement study and forensic analysis on real-world Dapp attacks, leveraging the open and immutable transaction records kept by the Ethereum blockchain to recover critical CTI. Particularly, our study has led to the discovery of a general, unique lifecycle of Dapp attacks, with the adversary showing similar behaviors in orchestrating attack operations against different target Dapps, regardless of low-level exploit techniques. Also we brought to light the adversary's attack infrastructures, campaigns they organized, as well as the inadequacy of the current response by defenders.
- We demonstrate that our new understanding and CTI discovered can help mitigate the threat to Dapps, using a new methodology developed for finding new attacks at different stages. Our approach leverages the similarity of attack behaviors exhibited by the transaction time series, which allows us to accurately capture both known and unknown attacks. This study shows that our findings could be leveraged to build a protection system down the road, to disrupt an exploit even before any damage has been caused.

2 Background

2.1 Ethereum and smart contract

Ethereum is a public blockchain-based distributed computing platform and operating system featuring scripting functionality. On the platform, there are two types of accounts: *Exter-*

nally Owned Accounts (EOAs) controlled by private keys (representing persons or external servers), and *Contract Accounts* controlled by code, which are known as *smart contracts*. The Ethereum blockchain [49] is the most prominent framework for smart contracts, where over 1 million contracts have been deployed [11].

Transaction. During its operations, the Ethereum blockchain tracks every account's state: once value has been transferred between accounts, the blockchain's state is also changed accordingly [27], which is recorded in a *transaction*. A transaction is a signed data package storing a message to be sent from an EOA to another account, which carries the following information: `to` (the recipient), `from` (sender's signature), `value` (the amount of money transferred from the sender to the recipient), `data` (the input for a contract), `gasprice` (the fee required to successfully conduct a transaction, i.e., `gas`, which is paid by the sender), etc. In Ethereum, all transactions are written onto a cryptographically-verified ledger [49], with a copy kept by every Ethereum client.

There are three types of transactions supported on Ethereum: Ether transfer, and contract call, contract creation [48]. The type of transactions can be determined based on the transaction format: an *Ether transfer transaction* transfers between two parties the amount of Ether as indicated by its *value* field; The *contract call transaction* is used to interact with an existing smart contract, with its `data` field specifying the method to call (e.g., the methodID of run() or kill()) and call arguments, and its `value` field carrying the amount of Ether to deposit in the contract (if the contract accepts Ether).

A *contract creation transaction* has its `to` field set to empty, and its *input data* field contains the bytecode of the contract. A typical bytecode is composed of the creation code, runtime code and swarm code, where the creation code determines the initial states of the contract, the runtime code indicates the functionality of the contract, and the swarm code is used for the deployment consistency proof and not for execution purpose. Typically, the creation code ends with the operation sequence: `PUSH 0x00, RETURN, STOP, 0x6000f3000`, and the swarm code begins with `LOG1 PUSH 6` in bytecode. This can be used to split the bytecode and identify the runtime code. In our research, we leveraged the contract creation transaction to recover the runtime code of the self-destructed contracts (Section 3.1).

Each executed transaction creates a receipt, keeping track of such information as the created contract address (`contractAddress`, as shown in Appendix Figure 11(e)) and the transaction execution status (0 for failure and 1 for success, as shown in the `status` field).

Smart contract concept and execution. A smart contract is used to facilitate, verify, and enforce the negotiation or performance of an agreement. As mentioned earlier, on Ethereum, such a contract can be created, executed and destructed by a *transaction* issued by an account. On reception of a transaction, a contract is run by the Ethereum Virtual Machine (EVM) on every node in the network. During the execution, the contract may communicate *internally* with other EOAs and contracts. Note that, to understand what data has been modified or what external contracts have been invoked, the transaction execution needs to be traced via re-executing a transaction under all historical states it accesses.

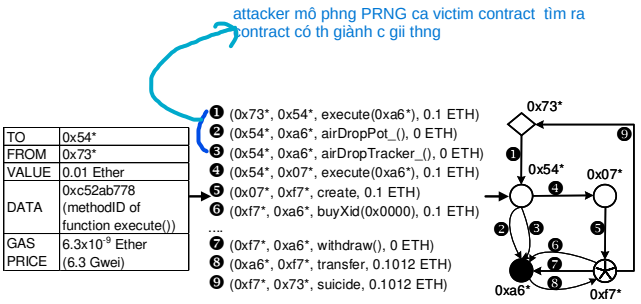


Figure 1: Example of transaction execution traces. ○: exploit contract, ⊛: contract generated in execution, ●: Dapp, ◇: EOA.

Figure 1 illustrates the execution traces (❶-❾) of a contract-call transaction, which is sent from 0x73* to call the function *execute()* of the contract 0x54* with a 0.01 ETH transfer. The transaction has triggered a set of execution traces, such as an internal call *airDropPot_()* from 0x54* to 0xa6* (❷), followed by another call to *airDropTracker_()* from 0x54* to 0xa6* (❸).

In our research, we model the set of the transaction's `execution traces` e_t at time t as a sequence of 4-tuples (I,O,B,T), i.e., $e = \{(I_i,O_i,B_i,T_i)|i = 1...n\}$, where I_i is the address triggering the behavior B_i (the function invoked and its parameters) on the recipient address O_i, together with a money transfer T_i (a transaction field recording the Ethers transferred from the issuer of the transaction to its recipient) at the step i.

In our study, we collected 11,960,145 execution traces of 2,350,779 transactions from Bloxy [13], and further constructed a directed and weighted graph for transaction analysis (Section 3).

2.2 Ethereum Dapps

Ethereum Dapps are public de-centralized applications that interact with the Ethereum blockchain, providing services such as gambling, online voting, token system, cryptocurrency exchange, etc. Such an application utilizes a set of smart contracts as its *on-chain* back-ends, for the purposes such as encoding task logic and maintaining persistent storage of its consensus-critical states [17], while also contains off-chain components such as its front-end (e.g., a website) for communicating with users. As an example, the Ethereum Dapp Fomo3D, a lottery game, is powered by a smart contract that handles the transactions for different actions, like buying keys, withdrawing from vault, picking a vanity name, etc. Note that in addition to acting as the back-end of a Dapp,

a smart contract can serve other purposes such as offering an on-chain library, and is also used to call a Dapp. In our research, we focus on the on-chain threats to the Dapp's back-end, a set of related contracts supporting the service of the application. These contracts are invoked by EOAs through other contracts or transactions. Below we also use the term "Dapp" to refer to the back-end smart contract(s) of a Dapp.

In our study, to identify Dapp among smart contracts, we utilize Dapp aggregation website [1] to recognize the Dapp names with their corresponding contract addresses and categories (e.g., gambling, game, finance, exchange). In this way, we identify 1,169 Dapps with 5,786 contract addresses and 18 categories. Note that Ethereum does not distinguish Dapp contract and non-Dapp contract naturally: if a Dapp has never been recorded by those websites, we cannot build the Dapp name-contract mapping.

Attacks on Dapps. As the largest Dapps market, Ethereum has seen quite a few high-impact real-world attacks on Dapps [36], resulting in losses of millions of dollars. Table 1 lists the types of attacks ever reported from 2016 to 2019 and the number of attack incidents. In our study, we utilize these published reports as seed to recover critical CTI on Ethereum Dapp attacks.

Here we present a real-world example of the Ethereum Dapp attack that exploits a weak randomness vulnerability in the *airdrop*() method of Fomo3D (see Table 1) for profit. Fomo3D is a highly-popular Ethereum gambling game with over 150,000 transactions a day and a prize pool of around $3 million in 2018 [42]. In the game, a player has a chance to win a prize from the *airdrop pot airDropPot_* when purchasing keys through *buyXid*(). More specifically, when *buyXid*() is being called, the Dapp first runs *isHuman*() to ensure that the caller is an EOA, not a contract, and then produces a random number through the pseudo-random number generator (PRNG) *airdrop*() to determine whether the player wins. The *airdrop*() method utilizes the parameters *airDropTracker_*, message sender address and block information (e.g., timestamp, difficulty, gaslimit, number, etc.) for generating pseudo-random number. During the attack, as shown in the execution traces of the exploit transaction in Figure 1, the attacker creates multiple contracts, e.g., 0xf7*(❺), from different message sender addresses. Since these contracts can get all parameters of the PRNG, they can implement their own *airdrop*() to find out whether they will win, and only the winning contract, e.g., 0xf7* (❽), purchases a key (❻). After that, the contract runs *suicide*() to transfer the prize to the attacker 0x73* (❾). Note that this attack circumvents the protection of *isHuman*(), buying a key through a contract instead of an EOA. This is because the implementation of *isHuman*() determines whether an address is an EOA or a contract from the size of the code associated with the address. This is unreliable since the contract under construction [40] could bypass the restriction (❹❺). We elaborate on this attack in Section 3.

2.3 Threat Model

In our research, we consider miscreants who launch attacks on Ethereum Dapps for profit. For this purpose, the miscreants could conduct several types of attacks on Dapps' contract vulnerabilities, such as exploiting weak randomness of a pseudo-random number generator (PRNG) in a gambling Dapp to win a prize, or performing integer overflow/underflow to manipulate money transfer, etc. We did not consider the attack in which the miscreants utilize a single EOA address to generate a single exploit transaction during the attack, which though possible, is rare in the wild (see Section 3).

3 Understanding Dapp Attacks in the Wild

In our analysis of Dapp attacks, we leveraged a variety of vantage points, including historical transactions and transaction execution traces, to reconstruct real-world Dapp attack incidents. Given the comprehensive transactions and their execution traces for each attack incident, we aim at identifying adversaries' end-to-end footprints and understanding their operational intents. Below we first describe the methodology we used to reconstruct the attack, and then elaborate on our findings and their security implications.

3.1 Data Collection and Derivation

Here we elaborate the design and implementation of a methodology that extends limited information collected from technical blogs and reports to tens of thousands of transactions related to Ethereum Dapp attack incidents (i.e., *exploit transactions*), and further analyzes the attack operations from these transactions. More specifically, our approach first reconstructs real-world Ethereum Dapp attack incidents, as documented by technical blogs, news posts, and the security reports from blockchain security companies, by recovering all transactions issued by attacker EOAs or exploit contracts, even when the transactions are not publicly disclosed. Then, to understand attack operations related to the exploit transactions, we model their fine-grained execution behaviors using their execution traces, and further determine their coarse-grained operational semantics by clustering the exploit transactions based upon the similarity and timings of their execution traces.

Exploit transaction collection. We first searched the Internet to collect real-world Ethereum Dapp attack incidents. In particular, we investigate three types of incident reporting sources, including technical blogs, news posts, and annual security reports from blockchain security companies. From these sources, we further manually picked out those related to Ethereum Dapp attacks. Details of these incident reports are presented in Table 14 in Appendix. Then, we reviewed these incident reports to identify immutable attack-related information (in the following called the seed attack set D_s), including victim Dapp addresses, exploit contract addresses, attacker EOAs, and exploit transaction hashes. In this way, we

Table 1: Real-world Dapp attacks

Attack type	Definition	# attack incidents
Bad randomness	adversary predicts the random value produced by the Dapp running a weak pseudo-random number generator (PRNG) to gain advantage (e.g., in a game)	6
Denial of service	adversary seeks to prevent legitimate invocations of a smart contract, through exhaustion of gas (constrained by block gas limit [41]) or improper check of exceptional conditions [47]	4
Integer overflow and underflow	an incorrect arithmetic operation that causes its result to exceed the maximum size of the integer type or go below its minimum value that can be represented	26
Reentrancy attack	a contract calls an external contract that unexpectedly calls back to the calling contract, rendering it operate in an inconsistent internal state [37]	2
Improper authentication	adversary exploits the authentication process that a Dapp uses to verify the ownership of resources, to enforce a behavioral workflow or to access a variable. It could be caused by typographical errors in contract implementation or missing protection on critical variables	15

Table 2: Known Dapp attacks. D_s is the set of data collected from the reports, and D_e includes those derived.

Attack type	# of Dapps		# of exploit contracts		# of attacker EOAs		# of attack transactions	
	D_s	D_e	D_s	D_e	D_s	D_e	D_s	D_e
Bad randomness	4	14	9	19	9	27	14	40,766
DoS	4	6	3	3	5	88	4	17,088
Integer overflow/underflow	13	32	1	2	28	53	47	591
Reentrancy	2	2	2	3	2	4	2	30
Improper authentication	12	18	6	18	17	60	34	575
Unique total	25	56	20	45	48	227	77	58,555

identified 42 Dapp attack incidents from 2016 to 2018, which consist of 25 victim Dapps, 20 exploit contract addresses, 48 attacker EOAs, and 77 exploit transaction hashes. Table 2 summarizes attack information we collected from the reports.

To reconstruct the reported incidents, we will look into all transactions, which were issued by attacker EOAs or exploit contracts to interact with the victim Dapps. However, such EOAs and exploit contracts may not be fully documented by the reports (see Table 2). Here we elaborate a methodology for finding the missing EOAs and exploit contracts.

First, to identify other EOAs in an attack incident, we include in the attack set all the EOAs that have created, called or transferred fund into known exploit contracts, or have transferred fund to known attacker EOAs. More specifically, we examine the transactions, whose `to` or `from` fields contain reported attacker EOAs or exploit contracts. Here we consider an address to be an EOA but not a contract if no code is associated with it. For this purpose, we use the function *w3.eth.getCode()* in python to get the size of the associated EVM code. A problem is that a self-destructed contract also reports a zero code size. In this case, to determine whether an address belongs to a self-destructed contract, we search for its creation transaction, the one whose `contractAddress` field contains that address (see Section 2).

Further we expand the seed attack set D_s by adding the contracts that are similar to the exploit one and have been called by attacker EOAs. More specifically, we extract the contract addresses, which were called by attacker EOAs, within a time window (1 day in our study) before and after the exploit transactions. Then we analyze the similarity of the extracted contracts and the exploit contract. In particular, we convert

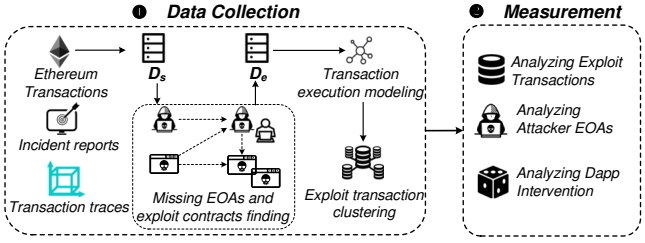


Figure 2: Workflow of the measurement approach.

the bytecodes into opcodes using Octopus [6], and then calculate their Jaccard similarity [29]. When they come close (Jaccard similarity ≥ 0.9), we consider them to be similar and the extracted one to be an exploit contract. Note that the adversary can use suicide operations or self-destructive operations to conceal his exploit contracts. In this case we recover the runtime code of a self-destructed contract from the contract's creation transaction (see Section 2).

In this way, we built an expanded dataset D_e, which contains 45 exploit contract addresses, and 227 attacker EOAs. We consider the exploit transactions to be (1) all those related to exploit contracts, and (2) those related to attack EOAs and issued within a 1-day window of a known exploit transaction. Altogether, we gathered 58,555 exploit transactions from 2016/01/29 to 2019/01/07, which involve in 56 victim Dapps (29 have never been reported before). To the best of our knowledge, this is the *largest* dataset for on-chain victim Dapp attack incidents that have ever been reported. We will release it on publishing this paper.

Transaction execution modeling. To understand attack op-

erations, we analyzed the executions triggered by the exploit transactions. In particular, we model a transaction's execution traces using a execution trace graph TG.

A transaction's execution trace graph TG is a directed and weighted graph as illustrated in Figure 1, in which each node is an account (i.e., EOA or contract address), and each directed and weighted edge describes an operation from one account to another.

Definition 1. A TG is a directed and attributed graph $TG = (V, E, W, t)$ in a node attribute space Ω, where:

1. V is a node set, with each node being an account (i.e., EOA or contract);
2. Each node is assigned one of five attribute labels in Ω: Dapp, EOA, self-destructed contract, Dapp related contract and other contract.
3. Directed and weighted edge set $E \subseteq V \times V \times W$ is a set of operations between accounts, where W is a set of call functions and parameters, e.g., *execute()* in Figure 5.
4. Time t is the timestamp of the transaction (when it is created).

Given a set of execution traces of a transaction $e = \{(I_i, O_i, B_i, T_i) | i = 1...n\}$ (see Section 2), an attribute graph TG can be constructed: here, the node set V is the collection of I_i and O_i, E is the set of edges from I_i to O_i if (I_i, O_i, B_i) exists with the edge weights of the call functions and parameters related to B_i. In our research, we gathered 436,371 execution traces for 58,555 transactions using Bloxy API [13] .

Exploit transaction clustering. To understand the semantics of the exploit transactions, for each attack incident, we clustered transactions based upon their execution traces' similarity and timings (within a given time window). This is essentially a between-graph clustering problem [9], which we solved using a k-Means algorithm and a TG distance.

Definition 2. A TG distance $D(g_1, g_2)$ is a distance between two transaction graphs g_1 and g_2 that measures both their structure similarity and timing closeness, as follows:

$$D(g_1, g_2) = \alpha \min_{(o_1,...,o_k) \in O(g_1,g_2)} \sum_{i=1}^{k} c(o_i) + \beta \Delta t \qquad (1)$$

where, $O(g_1, g_2)$ is a set of graph edits (e.g., vertex or edge's insertion, deletion and substitution) that transform g_1 to g_2, $c(o_i)$ is the cost for each edit, Δt is the time difference (with the unit of hour) between two graphs and α, β are the weights.

In our implementation, we used $\alpha = 0.9$, $\beta = 0.1$, $c() = 1$, adapted a python library GMatch4py [5] to compute D, and set the number of iterations for k-Means to 3. In Appendix 7.2, we present an analysis of the clustering performance and the discussion on the rationale for threshold selection. In this way, we gathered 126 transaction clusters related to 42 real-world Dapp attack incidents from 2016 to 2019.

Table 3: Reported contracts under different parameter settings (s: Jaccard Similarity; t: time window; TP: true positive)

Parameter	# reported contracts (TP)	Parameter	# reported contracts (TP)
s=0.9	45 (45)	t=1	45 (45)
s=0.7	86 (50)	t=3	58 (46)
s=0.5	126 (54)	t=5	77 (48)

Discussion. The aforementioned methodology can only serve as a measurement tool to derive exploit transactions and gain insight into the Dapp attack footprints, instead of a full-fledged detection system. Hence, to construct the expanded dataset D_e, we set the thresholds (i.e., time window, the Jaccard similarity of opcodes) for achieving a high precision, which might however miss some exploit transactions. To estimate the coverage, we lower down the threshold to improve the recall at the expense of precision to compare the findings with those reported with the original threshold.

Table 3 lists the number of reported contracts under different parameter settings of opcode Jaccard similarity and time windows. For the threshold of the similarity, when it is 0.9, we observe that all 45 reported contracts are indeed exploit contracts; when it becomes 0.5, our approach report 81 new contracts. We manually investigate all those newly-reported contracts and found only 9 exploit contracts (false negative), while the remaining 72 were all false positives, associated with 1,174 wrongly-reported transactions. Taking a close look at these 15 missing cases, we find that all of them are the evolved exploit contracts of the reported ones to optimize the functionality (Section 3.2).

Similarly, with the threshold of time windows increasing from 1 to 5, our approach report 32 more contracts associated with 127 transactions. After manually analyzing all newly-reported contracts, we found that only three are the exploit contracts (false negative), where the attacker took a long time interval (5 days) before using the same exploit contract to launch the attack on the same Dapp again. It might be because the attacker wants to test the original exploit on the patched Dapp.

3.2 Analyzing Exploit Transactions

Our data collection and derivation method reconstructs 42 real-world Dapp attack incidents, consisting of 126 semantic-similar transaction clusters with 58,555 transactions. Based on these transaction clusters, we manually annotated them and further performed a measurement study to understand the criminal footprints and operational intents of Dapp attacks.

Overview: attack footprints. Before coming to the details of our findings, we here first summarize the footprints of a typical Dapp attack discovered in our research, which consists of four stages: attack preparation, exploitation, attack propagation and mission completion, as illustrated in Figure 3. In the attack preparation stage, a Dapp attack starts

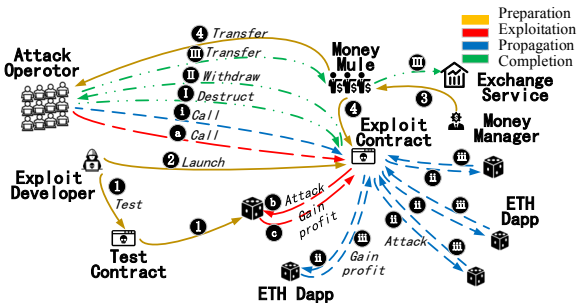


Figure 3: Example of Dapp criminal footprints, which consists of a four-stage attack lifecycle: attack preparation (❶-❹); exploitation (ⓐ -ⓒ); attack propagation (ⓘ -ⓝ) and mission completion (❶ -ⓜ).

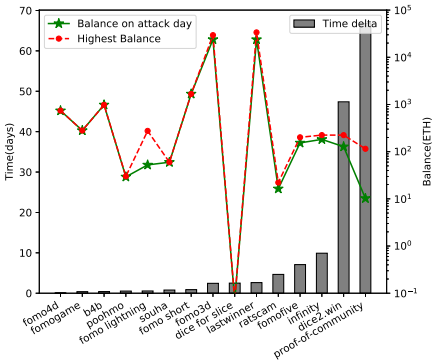


Figure 4: Balance of victim Dapps when miscreants started the attacks. The bar in the figure indicated the time difference between the Dapp launch time and the attack launch time.

with several transactions for calling the victim Dapp from *exploit developers* to test their exploit codes (❶) before the full attack is launched on the target (❷). Meanwhile, we observe several transactions through which *money managers* transfer attack cost (gas fee or ticket fee) into the exploit contracts (❸). This is done through *money mules* to conceal the managers' EOAs (❹). Then, in the exploitation stage, multiple *attack operators* from different EOAs invoke the exploit contracts (ⓐ) to attack the victim Dapp (ⓑ) and gain profit (ⓒ). After the attack, in the attack propagation stage, we found that the operators either reuse or further adjust the exploit contract (through update) (ⓘ) to exploit other similar Dapps (ⓝ) to gain more profit (ⓝ). During the mission completion stage, the attack operators destruct the exploit contracts (❶) and withdraw attack profit (ⓛ). The profit is then transferred from the attack operators or the exploit contract to the exchange service through several money mules (ⓜ). Below we elaborate on our measurement study and forensic analysis on real-world Dapp attacks.

Attack preparation. We first analyzed how the attacker bootstraps an attack. To this end, for each attack incident, we looked at all transaction clusters executed before the attacker continuously gains profit. More specifically, for each transaction, we evaluated whether the attacker profits by calculating the difference between his attack cost (i.e., money transferred from the attacker EOA or the exploit contract to the Dapp) and his attack gain (i.e., money transferred from the Dapp to the attacker EOA or the exploit contract). If the attacker continuously made profits from all of the transactions in a cluster, we considered that he has successfully launched an attack. Meanwhile, the clusters of the transactions executed before the attack were marked as being associated with attack preparation. In this way, we found the presence of such a preparation stage in 85% of attack incidents with the average number of transactions being 23. Also, the related preparation transactions were discovered within 81 days after the target Dapp was released. Surprisingly, we found that the weak randomness attacks were prepared in just 9 days after the appearance of the target Dapps. This might be because those Dapps usually share a similar vulnerable PRNG (see Section 2), and can thus be easily attacked once the PRNG has been exploited in one Dapp. Such attacks can be prepared by the miscreants once the target Dapp has some balance after launched (3461.5 Eths on average as observed in our study). Figure 4 illustrates the balance of the victim Dapps.

When manually investigating operational intents of the transactions in the preparation stage, we found that the attacker's transactions mainly serve two purposes: (1) testing their exploit contracts and (2) transferring fund to bootstrap their attacks (e.g., paying the gas fee). As an example, before attacking the vulnerabilities in the two Fomo3D functions *isHuman()* and *airdrop()* through an exploit contract (0x7d*), the attacker 0x85* created two test contracts 0x56* and 0x80* to evaluate these functions repeatedly. Apparently, the adversary performed his own *software integration testing* to ensure that all attack components worked smoothly together before executing the attack. In total, we found that 78% of the transactions at the attack preparation stage were used for such integration testing, with 8 testing contracts deployed and 96 transactions executed for this purpose in an attack incident. Furthermore, from the execution traces of these transactions, we identified 36 Dapp functions being tested. 79% of them were later attacked at the exploitation or the attack propagation stage. This indicates that by identifying the preparation stage, we could predict the vulnerable functions to be exploited and stop an attack before it occurs (see Section 4).

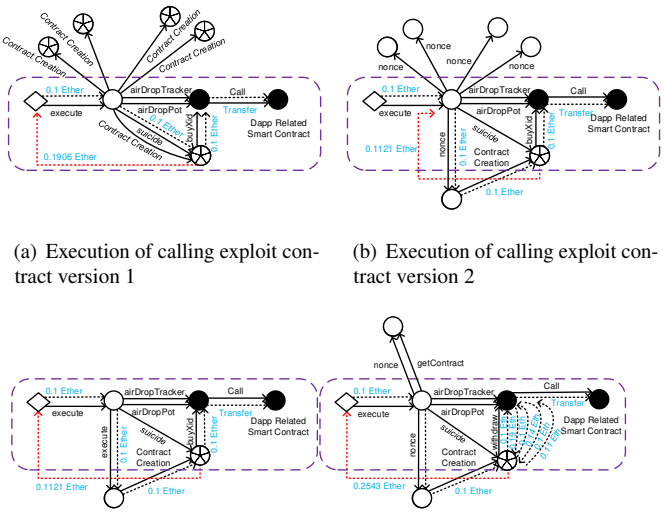
Attacks on Dapps come at a cost. For example, the attacker may need to purchase a ticket for playing a game Dapp before he can exploit its vulnerable functions, or pay a gas fee to launch exploit transactions. In our research, 324 transactions were discovered to transfer Ethers from EOAs or Ethereum exchange services to exploit contract addresses or attacker EOAs. As an example, in the attacks on Fomo3D, some attacker EOAs got inflows of Ethers from one EOA 0xbf*, through a set of intermediary EOAs (such as 0x2c*, 0xa7*

and 0x4c*) that were sequentially linked together to form money flow chains. Note that those intermediary EOAs associated with only two types of transactions, either receiving fund from a source or transferring it to another address. Although acting as a money mule, intermediary EOA shows a poor characteristic regarding anonymity, which is aligned with the findings in the Bitcoin laundry [20].

Exploitation. As mentioned before, we determine the transactions executed at the exploitation stage when the attacker continuously makes profits from one Dapp. On average 1,394 exploit transactions from 6 attacker EOAs were observed per incident. These transactions were used to either directly invoke vulnerable Dapp functions, or deploy or trigger an exploit contract to automate an attack. In total, we found from our dataset 232 transactions for calling vulnerable functions, and 22,269 transactions for triggering exploit contracts. Particularly, attacks on weak randomness and improper authentication, along with DoS, tend to utilize exploit contracts, since in these attacks, each exploit transaction call only brings in a small profit (e.g., prize per one guess), so the adversary needs to run an exploit contract to continuously invoke the target Dapp. On the other hand, in a reentrancy or an integer overflow/underflow attack, attacker EOAs usually directly exploit the vulnerable functions in the target.

To better understand the operational intents of the attackers at the exploitation stage, we analyzed the execution traces of their transactions. Of particular interest is the observation that the adversary tends to rapidly evolve his strategies during an attack, to improve its effectiveness (e.g., more revenue or less cost). Specifically, attackers were found to update their exploits via *delegatecall()*, or creating new contracts. For example, in the bad randomness attack on Fomo3D, as shown in Figure 5, we observed the presence of three exploit contract versions: since the *airdrop* function in Fomo3D heavily relies on the calling contract's block information (such as timestamp) to determine the winner, the first exploit version simply creates many new contracts to predict the function's output using the block information and the public logic of *airdrop* before invoking it; improving on the first version, the second one evaluates existing contracts' blocks through *nonce()*, and utilizes the contract on the winning block to generate a temporary contract (which still use its creator contract's block) to trigger *airdrop*, so as to save the cost for contract creation; the last version collects all information from existing contracts and makes the prediction off-chain before commanding the most promising contract to invoke *airdrop*. With the evolution, our research shows that the execution traces of these attack versions turn out to be similar (average TG distance = 0.4). This allows our tool *DEFIER* to uncover a new exploit version never reported before (Figure 5(d)).

Attack propagation. Given the existence of many copycat Dapps sharing the same vulnerabilities, our research shows that attackers tend to reuse their exploit on one target to infect



(a) Execution of calling exploit contract version 1

(b) Execution of calling exploit contract version 2

(c) Execution of calling exploit contract version 3

(d) Execution of calling exploit contract version 4

Figure 5: Exploit contract evolution at the exploitation stage. ○: exploit contract, ⊛: contract generated in execution, ●: Dapp, ◇: EOA. ——: function call flow; - - - -: data flow.

other similar Dapps. In particular, for the transaction clusters that ran after the exploitation stage, we discovered the transactions associated with the attack (i.e., continuously makes profits) but involved in the Dapps different from the one at the exploitation stage.

Looking into these transactions, we found that the adversary reuses his exploit contract through either creating a new contract with most its content copied from the old one or *delegatecall()* to invoke external code to run in the original contract's context. *delegatecall()* allows the adversary to simply adjust the external code to aim the exploit contact at different targets. For instance, at the propagation stage of the Fomo3D attack, the attacker EOA 0x82* deployed a new contract (the external code) to feed new vulnerable Dapp addresses to an existing exploit contract through *delegatecall()*. In this way, the attacker was able to reuse the exploit against 8 more Dapps simultaneously, including Fomo Lightning, Fomo Short, etc.

This *attack propagation* stage is found to come right after the exploitation stage, just 3.5 days apart on average. The bad randomness attack and integer overflow/underflow attack tend to have an aggressive propagation stage, with at least four more Dapps victimized per attack incident. For example, an integer overflow attack on Rocket Coin was propagated to another 17 Dapps.

Also we found that the adversary could scan Dapps' function names or runtime codes for the new targets carrying the same vulnerability as the victim Dapp. This is based upon the observation that 51% of the Dapps exploited at the propagation stage share the exactly same vulnerable function name or function bytecode with the Dapp attacked at the exploitation stage. Table 4 and 5 list the functions and the variables

Table 4: List of vulnerable functions

Functions	#Dapp	Attack type	Jaccard sim.
transferFrom	16	Integer overflow/underflow	0.64
airDrop	8	Bad randomness	0.99
transfer	7	Integer overflow/underflow	0.78
transferProxy	6	Integer overflow/underflow	0.83
batchTransfer	5	Integer overflow/underflow	0.82

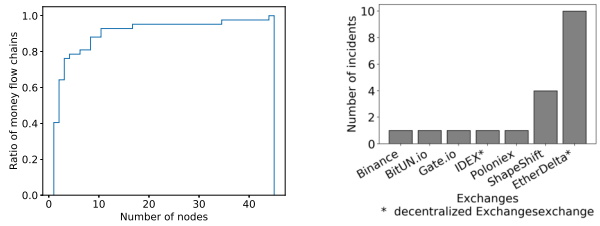
Table 5: List of vulnerable variables.

Function	Vulnerable variable	# attacks
transferFrom	value	16
airDrop	airDropPot	8
	airDropTracker	8
transfer	value	7
transferProxy	value	6
	v	6
	r	6
	s	6
batchTransfer	value	5

(under a given function) most commonly appearing in the attack incidents we collected. In particular, we observed that the function *transferFrom()*, which is used for transferring tokens between accounts, was exploited by the same integer overflow attack in 16 different Dapps.

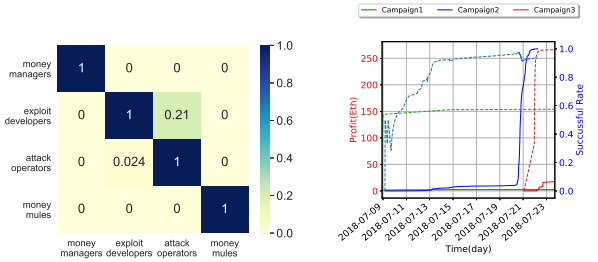
Mission completion. After a successful attack, our research shows that the attacker often withdraws all the profits he made and tries to remove attack traces by destructing all his exploit contracts. Specifically, our dataset includes the transactions to destruct exploit contracts by calling *selfdestruct()* or custom destruct functions. Actually, 35.6% of the exploit contracts in all attack incidents we studied were destroyed. Note that the destruction of a contract automatically transfers its winnings to the contract's creator EOAs.

Interestingly, once an EOA receives the fund from its contract, it tends to further transfer the winnings to another address. In our study, we identified 198 transfer transactions at the mission completion stage, and constructed the money flow chains on them in the same way as did when analyzing attack preparation. Figure 6(a) shows the cumulative distribution of the nodes in the money flow chains. We found that in 19% of money flow chains, illegal profit was transferred via at least one money mule. Also intriguing is the observation that the adversary always converts Ethereum tokens (e.g., Beauty Coin, Smart Coin, SmartMesh Token) into Ethers before moving the fund into a long money flow chain, possibly due to the belief that the latter have better protected values than the former. For this purpose, a set of Ethereum Exchanges are used. Figure 6(b) illustrates seven Ethereum Exchanges discovered from our dataset. There are two types of exchange services in Ethereum: centralized Exchanges (e.g., ShapeShift, Binance, Poloniex, Gate.io and BitUN.io) and decentralized Exchanges (e.g., EtherDelta and IDEX). From the data we collected, apparently miscreants are more in favor of the decentralized ones. Particularly, EtherDelta shows up in 53%



(a) CDF of the money flow chains (b) Exchange distribution

Figure 6: Mission completion.



(a) Role overlap of attacker EOAs (b) Profits of three Fomo3D campaigns. ——: successful rate; - - -: profit.

Figure 7: Dapp attack incident analysis.

of the attack incidents, while ShapeShift, the most popular one among centralized Exchanges, is just found in 21% of the incidents.

3.3 Analyzing Attacker EOAs

Then we looked into the role and relationships of 227 attacker EOAs discovered in our study. Our study shows that attacker EOAs are organized through a hierarchical structure during an attack incident, with each of them playing one or more roles. Further revealed in our study is the competition relation among different attacker EOAs when exploiting the same Dapp, across different attack incidents, as elaborated below.

Roles in an incident. We analyzed the roles of 227 attacker EOAs by first categorizing them based on the attack stages (Section 3.2) at which they appeared and then manually investigating their transactions to understand their behaviors. More specifically, we observed that 19 EOAs acted as exploit developers which created and tested exploit contracts at the first stage (see Section 3.2); 168 EOAs invoked exploit contracts or ran other exploit code, thereby likely playing the role of attack operators; further 21 EOAs apparently managed the attack cost inflow through transferring attack cost into the exploit contracts via intermediary EOAs, behaving like money managers, and 23 EOAs were found to relay attack profits, as money mules did.

Our study shows that attacks on Dapps are organized through a hierarchical structure in which every actor has a well-defined role. There is only a small overlap among different roles: Figure 7(a) shows that rarely do we see that an EOA

played more than one role, except that 21% of the exploit developers also acted as attack operators.

Campaign competition. As mentioned earlier, 39% of the victim Dapps have been exploited in more than one attack incident. Interestingly, our research reveals the presence of competitions among different attack campaigns on the same Dapp. Here, a campaign is considered to include all attacker EOAs showing up in an attack incident against a target Dapp.

Figure 7(b) compares the cumulative attack profits of three campaigns on Fomo3D from 2018/06/15 to 2018/08/31. Each of them involved a completely different set of EOAs from others and therefore presumably they were organized by different parties. Campaign 1 first launched a bad randomness attack on the Dapp on 2018/06/15, followed by Campaign 2 on 2018/07/08 and Campaign 3 on 2018/07/21. Here we use the exploit success rate, defined as the number of successful exploit transactions (i.e., receipt status is 1) among all exploit transactions, to measure attack effectiveness. Although starting relatively late, Campaign 2 evolved its exploit contract on 2018/7/20 to increase its effectiveness. Hence, it made more profits than the other two campaigns. For Campaign 3, even though it apparently was quite effective (see Figure 7(b)), the attack only lasted for a short period of time and earned only a small amount of profit, probably due to the fact that Fomo3D had already lost most of its money during the attack.

3.4 Analyzing Dapp Intervention

We further studied how Dapp owners responded to the attack incidents by analyzing Dapp's transactions after an attack occurs. We observe some Dapp owners abandoned their Dapps (33 out of 56 victim Dapps), while others tried to fight back, through patching, hiding source code or controlling access to the critical functions. None of them, however, is found to be a perfect solution in our research.

Dapp patching. Patching a vulnerable Dapp is complicated due to the immutability of the code stored on the blockchain. A typical solution is to create a new contract with the patch. To understand this procedure, we extracted Dapp's original addresses from its website's archive. We found that five of the Dapps analyzed in our research updated their contract addresses after being attacked, and one used *delegatecall()* for patching. Interestingly, three Dapps were attacked again after patching. For instance, Lucky Blocks changed its address twice to fix vulnerabilities yet still ending up being exploited.

Closed source. Another way is security by obscurity, hiding source code in an attempt to raise the challenge in reverse-engineering. A prominent example is Lucky Blocks, a gambling game, whose source code was removed right after a bad randomness attack. Indeed, we did not see any more attack on the Dapp after that. This approach, however, could make some Dapp less trustworthy. Again, for Lucky Blocks, through analyzing its PRNG in the patched version, we discovered that the Dapp owner stealthily adjusted the code to limit the range

```
function getRandom() returns (var r0) {
  ...
  var temp0 = memory[0x40:0x60];
  memory[temp0:temp0 + 0x20] = block.difficulty;
  ...
  return keccak256(memory[temp1:temp1 + temp0
- temp1 + 0x54]) % 0x64; //0x64=100}
```

(a)

```
function getRandom() returns (var r0) {
  ...
  var var1 = 0x5c; //92
  var temp0 = memory[0x40:0x60];
  memory[temp0:temp0 + 0x20] = block.difficulty;
  ...
  var var2=keccak256(memory[temp3:temp3+(temp2+0x20)-temp3]);
  if (var1){return var2 % var1;} else {assert();}}
```

(b)

Figure 8: PRNG codes of Lucky Blocks.

of the randomly-produced lucky number, thereby reducing the winning chance by 8% (Figure 8). The Dapp later indeed shows higher owner-side revenue.

Administrator list. Finally, we found that 33 of the 56 victim Dapps utilized administrator lists to restrict access to their critical functions. However, the administrator list cannot stop the attack that exploits the vulnerabilities in an authentication mechanism to bypass access control. An example is the attack on Morph [45]. Also, this strategy requires the identification of critical functions beforehand.

4 Finding New Attacks

In this section, we show how the new CTI discovered can help find new attacks, including those on 0-day victim Dapps. Our key insight is that even though specific operations may vary across different types of attacks on different Dapps, the high-level behavior patterns (e.g., testing exploit contracts) are relatively stable in each attack stage (e.g., attack preparation stage), and can therefore be learned from a set of transactions and their execution traces. Here we elaborate on a methodology, called *DEFIER*, that utilizes the sequence of transactions and the operations they trigger to recover attack footprints and determine the stage of an exploit.

4.1 DEFIER: Idea and Design

DEFIER includes two components, *Preprocessing* and *Sequence-based Classification*. *Preprocessing* takes as its input a set of transactions directly interacting with a Dapp, automatically extending the set to include those indirectly related to the Dapp (Section 4.2). These transactions are then clustered into groups based on the similarity of their execution traces and the closeness in their invocation times (within a short window). These transaction groups are then utilized by *Sequence-based Classification* to re-construct potential attack footprints, in terms of a transaction sequence from multiple

EOAs (Section 4.3). More specifically, for each sequence of transactions (modeled as vectors through graph embedding), we propose a novel embedding technique to convert the sequence into a feature vector that captures the latent intent of the sequence (through an attention model to focus on each transaction's interactions with the Dapp and an analysis on the relation between transactions). Those vectors then go through a multi-class classifier to output the attack stage they belong to if they are indeed exploit attempts

Example. To explain how *DEFIER* works, here we walk through its workflow using an attack incident on Suoha, a victim Dapp found at the propagation stage of a bad randomness attack on Fomo3D. To investigate this attack incident, *DEFIER* identifies the latent intent (i.e., exploit calling at the propagation stage) by (1) clustering similar transactions from EOAs across different Dapps (e.g., transactions that launch the same exploit on multiple Dapps) and (2) then analyzing those transactions to find the latent intent.

More specifically, *DEFIER* first runs *Preprocessing* to gather transactions, whose `to` fields or execution traces contain Suoha's address. From those transactions, 286 EOAs (including those calling 7 contracts to interact with the Dapp) are extracted. Further, we gather the EOAs' transactions with other Dapps, those with a small TG distance with the transactions with Suoha. In this way, 11,088 transactions are identified and further clustered into 142 groups with an average TG edit distance of 0.2 and a time window of 1.5 hours. For each of these groups, *Sequence-based Classification* first runs graph embedding to convert each transaction to a vector and each group to a vector sequence and then utilizes an LSTM model to analyze the relation between the vectors in the sequence, converting each sequence to a feature vector. Then, a multilayer perception (MLP) classifier, trained over the transactions from reported attacks, labels 3 of the sequences as attack propagation and the remaining 139 as legitimate.

4.2 Preprocessing

The *Preprocessing* step is meant to gather and cluster relevant transactions to analyze *all* EOAs' operations and their intents on a Dapp. Such intents sometimes cannot be profiled only by the transactions directly interacting with the Dapp. For example, one can only recognize the intent to reuse exploit code on other Dapps (the propagation stage) by looking at the transactions on other targets, which look similar to the exploits on the Dapp (Section 3). Hence in our research, we include all such similar transactions, even though they are not directly related to the Dapp. Altogether, we consider the following two types of transactions during preprocessing:

• *Dapp transactions*. We collect the transactions with the Dapp and those that *internally* communicate with the Dapp (the transactions do not have the target Dapp address in their `To` fields but invoke its functions as discovered from their traces). For this purpose, our implementation relies on APIs *get_normal_txs_by_address* [3] and *get_internal_txs_by_address* [4] from Etherscan [2] to identify those transactions.

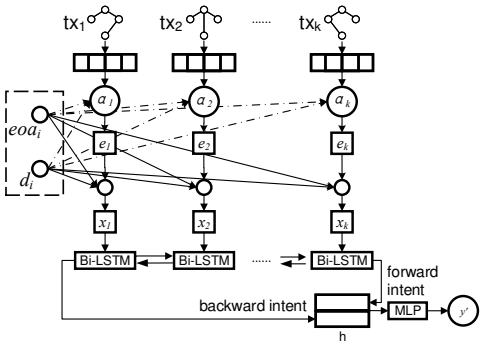


Figure 9: Sequence representation

• *Semantically-similar transactions*. Given those Dapp transactions, to better understand the operational intents of an EOA, we also gather from the same EOA the transactions with similar execution traces or occurring concurrently.

Specifically, we first identify all the EOAs directly interacting with the Dapp, including the addresses directly calling the Dapp and the ones creating a contract to invoke the call. To this end, we fetch the transactions whose `to` fields or execution traces contain the Dapp addresses, to identify a set of EOAs and contracts. Then, given a contract S interacting with the Dapp via a transaction tx_s, we collect all the EOAs who have created, called or transferred money into the contract S. In this way, we discover all relevant EOAs, which allows us to use the transactions to profile the behaviors of each EOA.

Such profiling is done by running Algorithm 1 on semantically similar transactions. In particular, given an EOA u interacting with the Dapp via a transaction tx_s, we acquire all her transactions whose TG distances with tx_s are within th (a threshold). In our implementation, we set th to 3 based on an empirical study (Section 3).

Transaction clustering. As mentioned in Section 3, an operational intent (e.g., exploit testing, multiple-step game playing operations) sometimes consists of several transactions from multiple EOAs. To find the transaction clusters under the same operational context, we utilize the algorithm described in Section 3 to group the transactions with similar execution traces or happened within a small time period.

Account de-noising. Complicating our analysis effort is the presence of *Dapp owner EOAs* and *library contracts* (e.g., a game playerbook contract for managing players' information or a contract supporting access to external network data), which should not be included in an attack investigation. To remove the noise, we first identify the library contracts through a Dapp's call execution traces: those invoked proactively by the Dapp are considered to be library contracts. For this purpose, we find all the contracts recorded by the call execution trace, whose "from" fields are the Dapp address and input fields are not "0x". To handle the library contracts, which had

Algorithm 1: Transactions Extension Algorithm

```
Data: Dapp: a dapp and its addresses.
begin
    EOAs = extract_eoa_of_dapp(Dapp)
    interval = 1 day
    threshold = 3
    for EOA ∈ EOAs do
        txs = get_txs_by_DappandEoa(Dapp, EOA)
        for tx ∈ txs do
            date = tx_date(tx)
            focus_period = calculate_period(date, interval)
            extend_txs = get_tx_in_period(EOA, date_period)
            picked_txs = [etx for etx in extend_txs if distance(tx,
              etx) ≤ threshold]
            save(picked_txs)
        end for
    end for
end
```

not been proactively called yet, we conduct a static analysis on the bytecode of a Dapp. In particular, we decompile the bytecode using [7], and then extract the library contract addresses using a regex "0x[a-fA-F0-9]{40}". Also, we retrieve Dapp creation transaction receipts (i.e., the receipts containing the `contractAddress` field of the Dapp address, which have been collected during the library contracts extraction) to extract the Dapp creator addresses from the `from` field.

4.3 Sequence-based Classification

From each transaction cluster, we form a *transaction sequence*, with transactions ordered by their timestamps. For a transaction sequence, we determine whether it describes an attack on a Dapp by predicting its latent intent (e.g., exploit testing, attack propagation, etc.) based upon the knowledge about other sequences with similar semantics. A semantically-similar transaction sequence $\hat{s}$ related to a Dapp attack stage y is represented as 2-tuple $(\{tx_i|i=1...k\},y)$, where $\{tx_i|i=1...k\}$ are transactions in $\hat{s}$ and y is the label of an attack stage. The goal of the sequence-based classification is to find the class label y for an input sequence $\hat{s}$ given the classifier's model parameters θ, i.e., $y' = argmax\ Pr(y|\hat{s},\theta)$, where the parameters are learnt from a training dataset. For this purpose, we first convert the transaction sequence $\hat{s}$ into a vector sequence, with each element also being a vector that represents its corresponding transaction graph through a graph embedding. This sequence is then fed to an LSTM model to generate a vector h that describes the relation between transactions and highlights the information related to malicious behavior. Here, we choose LSTM, a modified RNN, since it is designed to learn the long-term dependency relations among the elements on a sequence [28], which is critical for identifying the patterns that link transactions together at different attack stages. The vector is later classified by a multilayer perceptron (MLP) to determine whether it is indeed related to an attack stage.

Sequence representation. As illustrated in Figure 9, each transaction tx_i in $\hat{s}$, as described by its associated execution traces tg_i, represents an interaction between the corresponding Dapp and EOA. However, the transaction's execution can be too Dapp-specific and noisy to capture the operational intent, since the execution trace may contain many operations that happen inside the Dapp, for example, invocation of the Dapp's internal libraries to generate a pseudo-random number (Figure 5), which is less relevant to the EOA-Dapp interactions of interest to us (attack preparation, exploitation, propagation and completion). To address this issue, we employ an EOA-Dapp-execution attention model to highlight the useful information related to the EOA's intent on the Dapp. Here the attention a_i is used to adjust the vector representation of the transaction graph tg_i. It is determined by a weighted combination of the vector representations of EOA eoa_i, Dapp d_i (produced by a vertex embedding [25]) and that of tg_i (produced by graph embedding [19]). Its weights are learnt through an LSTM model (Figure 9) within an end-to-end deep neural network that ultimately outputs the feature vector characterizing whole input (the vector sequence representing a transaction sequence).

$$\begin{aligned} a_i &= softmax(NE(eoa_i) \oplus NE(d_i) \cdot GE(tg_i)^T), \\ e_i &= a_i \cdot GE(tg_i) \end{aligned} \tag{2}$$

where $\oplus$ is the concatenate operation, $NE()$ is the vertex embedding (e.g., node2vec [25]) of the input, which generates a vector representation for each node, $GE()$ is the graph embedding (e.g., structure2vec [19]) of the input, which generates a vector representation for each transaction graph, and $softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j))}$. In our implementation, the length of the node embedding is set to 64. We construct the concatenation of the EOA and the Dapp vertex embedding into a vector with a length of 128.

In the deep neural network, we further utilize a standard combination gate [28] to determine how much information from the EOA, the Dapp and the transaction execution will be used through adjusting their weights. In this way, we obtain the representation x_i of the transaction tx_i:

$$\begin{aligned} c_i &= \sigma(W \cdot (NE(eoa_i) \oplus NE(d_i) \oplus e_i)^T + b), \\ x_i &= (1 - c_i) \circ e_i + c_i \circ (GE(tg_i)) \end{aligned} \tag{3}$$

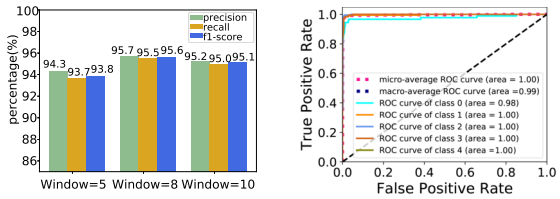
where W is a weight matrix, b is a bias, σ is the sigmoid function, and $\circ$ is the element-wise multiplication. Given transaction encoding x_i, we use a Bidirectional LSTM [24], which has been trained with the classifier (see below) on labeled dataset (Section 4.4), to capture the inner relationship between transactions. In this way, the transaction sequence can be converted into a vector h by the trained model.

Sequence classification. The output of the attention model, h, serves as the input to a multilayer perception (MLP) classifier. The MLP is used by *DEFIER* to generate the probability y' for a given attack stage the sequence is associated with. The

Table 6: Dataset and evaluation results.

Dataset	# transactions	Results
Groundtruth set	badset 57,855 goodset 39,124	pre_{micro} 98.2%, pre_{macro} 92.4% rec_{micro} 98.1%, rec_{macro} 98.4%
Unknown set	2,350,779	*positive* 476,334
Sampled testset	30,888	pre_{micro} 91.7% pre_{macro} 83.6%

pre_{micro} and pre_{macro}: micro of precision, macro of precision
rec_{micro} and pre_{macro}: micro of recall, macro of recall
positive: transactions that labeled as one of attack stages



(a) Model performance with different window size

(b) ROC

Figure 10: Evaluation results.

whole *Sequence-based classification* module, including the LSTM and the MLP, can be trained together through stochastic gradient descent, a typical way to train such a complicated model [14], on labeled data (Section 4.4). In our study, we built a Bi-LSTM with three folds, whose convolution sizes were 128, hidden sizes were 256 and batch sizes were 128. The epochs were set as 20 and learning rate was set as 0.0001. The hidden size of MLP was set as 256.

4.4 Evaluation

Here we evaluate *DEFIER* and elaborate on the challenges in multi-stage exploit transaction identification.

Evaluation with groundtruth set. We evaluated *DEFIER* over the following *ground-truth dataset* as shown in Table 6: for the bad set, we collected 57,855 transaction sequences associated with Dapp attacks from our measurement study. In particular, for exploit transactions in the same attack stage, we first order them by timestamp, and then define a sliding context window with the size of w (w=8 in our implementation) to chunk the time-ordered transactions into transaction sequences. Finally, we label those transaction sequences by their attack stages. We detail the annotation process in Appendix 7.3. In this way, we built a bad set with 57,855 transactions (469 at the attack preparation stage, 22,333 at the exploit stage, 34,763 at the attack propagation stage and 290 at the mission completion stage). The transactions of good set were gathered from 56 victim Dapps related to the bad set and 318 manually checked normal EOAs on these Dapps. Specifically, we ran the module of *Preprocessing* to generate the transaction sequences with the same size of context window. In this way, we construct a good set with 39,124 normal transaction sequences. Running on these sets under 10-fold cross validation, *DEFIER* shows a micro-precision of 98.2%, a macro-precision of 92.4%, a micro-recall of 98.1% and a macro-recall of 98.4%.

Table 7: Performance comparison in different models

Method	Attention	precision	recall	F1
RNN	no attention	0.965	0.962	0.963
RNN	attention	0.974	0.969	0.971
LSTM	no attention	0.977	0.975	0.976
LSTM	attention	0.982	0.981	0.981

Missed cases. On the ground-truth dataset, seven cases were missed by *DEFIER*. These transactions fell through the cracks due to inadequate attack-related semantic content in their clusters. In three cases, we found that the size of the sliding context window is not large enough to capture some attack behaviors, and as a result, the adversary's operational intents and the attack stages could not be determined. In other cases, the problem comes from the presence of reverted transactions, whose original execution traces cannot be obtained, which prevents *DEFIER* from building up their transaction graphs.

Determining the number of missed malicious transactions in the large-scale unknown set (with more than 2.3 million transactions, 342K clusters) is challenging. What we did in our research was to flag a transaction cluster as the class types with the largest predicted probability, as well as the second largest predicted probability when it is greater than 0.5. This strategy will include more flagged cases, at the expense of precision. In this way, our approach flagged 1,069 more transaction clusters. We manually analyzed all of them and found 167 new exploit transaction clusters. Looking into these missed cases, we found that 146 cases were caused by the window size or reverted transactions, as mentioned above. The remaining 21 cases resulted from the lack of Dapp information for transaction graph node labeling (see Section 3.1). This problem can be handled by a more comprehensive Dapp list.

Falsely detected cases. We also found two major causes for the 322 false positives observed in our study (Section 4.5). Those transaction clusters are either semantically similar to the clusters in another attack stage, or having attack patterns of multiple attack stages. For example, when attackers evolve their attack strategy (Section 3.2) frequently without exploitation behavior, our model may misclassify these exploitation clusters as attack preparation clusters. This is because the transactions during attack strategy evolution can be semantically similar to those for attack preparation: the adversary kept using new exploit contracts to interact with a Dapp, and attack costs were transferred to the new exploit contract to bootstrap attack. The second type of false positives is caused by the incorrect transaction clustering. For example, one transaction cluster of CityMayor consists of the transactions at attack preparation stage and exploitation stage, because the time interval between these transactions is small ($\leq$ 10 minutes), and the similarity of these transactions is large (average TG distance $\leq$ 0.33).

Table 8: Performance comparison in different epochs

Epoch	$learning_{rate}$	precision	recall	F1
10	0.001	0.965	0.962	0.963
20	0.001	0.982	0.981	0.981
50	0.001	0.980	0.980	0.980
100	0.001	0.994	0.980	0.980

Table 9: Performance comparison in different learning rates

Epoch	$learning_{rate}$	precision	recall	F1
20	0.1	0.958	0.914	0.932
20	0.01	0.978	0.977	0.977
20	0.001	0.982	0.981	0.981
20	0.0001	0.985	0.982	0.983
20	0.00001	0.918	0.906	0.908

Parameter and model selection. In Section 4.2, the size of the sliding context window w controls the length of the transactions used to inspect the operational context. Here a small window size might contain inadequate information about the operational context, while a large window may bring in the information across different stages, which leads to noise. In our research, we analyzed the impact of various w (5, 8, 10), as illustrated in Figure 10(a) and 10(b), over the ground-truth dataset, and chose the one with the best performance ($w = 8$).

Parameters such as the number of epochs and the learning rates for the LSTM model are used to control the performance of the model. In our study, we tuned the model by varying the number of epochs from 10 to 100 and the learning rates from 0.00001 to 0.1. From the result shown in Table 8 and Table 9, we can see that the classification performs best under 20 epochs and a learning rate of 0.0001.

In our study, we compared the effectiveness of RNN and LSTM models on the sequence classification tasks. Specifically, we implemented four models: RNN, RNN with attention, LSTM, LSTM with attention on the groundtruth dataset and evaluated their effectiveness using 10-fold cross validation. Similar to the LSTM model we used (Section 4.3), the backbone of the RNN is also three layers 128 * 256 * 128 with the batch size of 128. Table 7 shows the results. We observe that the LSTM with attention outperforms other sequence classification models.

4.5 Discovery and New Findings

We collected 104 popular Dapps and their corresponding contract addresses from a Dapp ranking list [8]. On these Dapps, we ran the *Preprocessing* to gather 2,350,779 transactions from Ethereum and construct 342,224 transaction clusters. *Note that we eliminate all the transactions used in the measurement study (Section 3). DEFIER* inspected these transactions and labeled 476,342 of them (100,081 clusters) with one of the attack stages. These transactions are related to attacks on 85 victim Dapps. For each victim Dapp, we randomly sampled 4% of the reported transaction clusters for manual validation. In total, we manually investigated 4,003 transaction clusters with 30,888 transactions. We found that 3,671 clusters are indeed related to attack incidents and 3,347 clusters are at the right attack stage.

Table 10: Victim Dapps in different categories.

Type	# Dapps/0-day	# attacker EOAs/0-day	# exploit transactions/0-day	ex. of victim Dapps
Gambling	51/43	65,778 /11,339	360,524 /114,473	Lucky Blocks
Game	28/27	959/919	52,673 /52,176	SpaceWar
Finance	5/5	183/183	59,872 /59,872	STOX
Token	2/1	279/167	4,478/472	Power of Bubble
Total	85/75	67,199 /12,608	476,342 /226,763	

Table 11: Unknown set result.

Attack stage	# Dapps/0-day	# attacker EOAs/0-day	# exploit transactions/0-day
Attack preparation	80/70	42,661/8,237	214,408/106,436
Exploitation	85/75	35,955/3,650	143,179/39,908
Attack propagation	75/65	18,466/6,545	118,755/80,419

Table 10 summarizes our findings. Our study reveals that Ethereum Dapps attacks are indeed prevalent, compromising various kinds of Dapps through different attack vectors. We observe that 57.3% of the victim Dapps are in the category of *Gambling*. To support the gambling functionality, these Dapps need to generate random numbers, which sometimes are implemented by a weak PRNG, thereby exposing the Dapps to the bad randomness attack. Note that in our study, 82% of the Dapps scanned by *DEFIER* were observed under attacks. This might be because the Dapps we analyzed were highly popular with large balances, which makes them more likely to be targeted by the miscreants. Also, among the 85 victim Dapps found in the exploit transactions, 75 (e.g., SpaceWar and SuperCard) were never reported before.

To understand the economic impacts of these abusive activities, we estimate the financial loss of the victim Dapp. In particular, for each victim Dapp, we calculate its income and cost difference of the exploit transactions. Table 12 shows the victim Dapps with the top-5 largest losses. The total loss inflicted by the attacks on these five Dapp is estimated to be 28,485 Ethers.

Table 11 shows the number of Dapps found in each of the attack stages. Interestingly, our model identifies 214,408 attack preparation transactions associated with 80 Dapps. We found 507 functions were tested by the adversaries. Interestingly, 311 functions were indeed exploited in the exploitation stage. It indicates that our model can help identify the vulnerable functions before they are exploited.

Table 12: Top-5 victim Dapps with largest losses.

Dapp	# transactions	# exploit transactions	Revenue (Eth)
LastWinner	561,845	101,304	13,295.2
Fomo3D	438,062	83,833	14,630.9
Dice2Win	69,874	8,919	185.0
Fomo Short	52,431	4,075	314.7
SuperCard	43,897	6,315	59.2

5 Discussion

Mitigation. Based on the results of our measurement study, we have identified several potentially effective mitigation strategies to control the fast-growing Ethereum Dapp attacks. In our study, we observed several stakeholders (e.g., exploit developer and money manager) in the Ethereum Dapp criminal ecosystem. Identifying such upstream criminal roles and monitoring or even restricting their activities (e.g., blocking them from accessing Dapps) could prevent attacks at the early stage (see Section 3.2).

Also, for the Dapp owner, an effective way to mitigate the threats she is facing is to detect an exploit attempt at its preparation stage, and also keep track of the exploits on similar Dapps to prevent the propagation attack. Particularly, since *DEFIER* identifies each stage of the kill chain without depending on other stages' information, it can be utilized for the attack preparation investigation. Also, as mentioned in Section 4.5, we found that 62% of the functions tested by the attackers at the preparation stage were indeed exploited later. Identifying these functions would help the Dapp owner to locate the vulnerabilities in her Dapp. In addition, our study reveals the prevalence of the attack propagation stage, in which attackers reuse their exploit on one target against other similar Dapps. Therefore, to prevent the attack propagation, the owner can use *DEFIER* for exploitation monitoring on her Dapps with similar functionalities and take actions before attacks happen.

Limitation of *DEFIER*. Our design is limited by the information it uses: historical transactions and their execution traces. Although these transactions provide valuable sources for attack investigation, they miss the attack operations that do not generate transactions, such as conducting a local invocation (e.g., `eth_call`) or calling a constant function of a Dapp (e.g., `constant`, `view` and `pure`). While those operations are read-only or do not change the Dapp state, and thus are found to be rarely exploited in the attack incidents (see Table 14), we acknowledge that our vantage point might cause some attack cases to fall through the cracks. We will leave a further study on the problem to the future research.

Also, as a supervised learning model, *DEFIER* required training set which labels transactions by its attack lifecycle. While we believe our paper yields meaningful CTI implications, which help data annotation, we acknowledge that the data annotation for our model can be time-consuming. However, since the training set aims at capturing high-level and relatively-stable attack intents, the training set can be used until those criminal intents change.

The design of *DEFIER* is based upon high-level threat intelligence (e.g., kill chain and attack patterns) instead of fine-grained Dapp-specific attack operations, and therefore is robust to the small adjustments of attack activities. However, the attack that does not exhibit the intent related to the stages or just involves a single exploit transaction with limited profit may not be identified. On the other hand, *DEFIER* would raise the bar to Dapp attacks, making them more costly especially to the adversary who wants to launch the attack on a large scale to make a profit.

Other blockchain platforms. Our current design is focused on Ethereum Dapps due to their popularity. However, such criminal operation mode can also be found in other blockchain platforms (e.g., EOS). In particular, we conducted a small-scale study on the attack incidents of EOS Dapps (i.e., EOS.WIN, EOSCast and EOSRoyale) and discovered a similar attack lifecycle and attack patterns from the EOS transactions and their corresponding execution traces.

6 Related Work

Study on Ethereum Dapp security. The security issue on Ethereum Dapp is attracting increasing attention from researchers. Aside from vulnerability assessment [16, 30, 50], studies on real-world Ethereum Dapp attacks and frauds are also conducted to understand the cybercriminal situation on Ethereum Dapps. For example, Chen et al. [16] studied the Ponzi scheme Dapps on Ethereum and built a machine learning based Ponzi scheme Dapp detection tool. Torres et al. [46] investigated another fraud Ethereum Dapps: honeypot, where attackers lure victims into vulnerable contracts. The paper introduced a methodology that uses symbolic execution for the automated detection of honeypot contracts. Chen et al. [15] identified abnormal EOA, that creates lots of contracts that are rarely used, by a threshold-based method. This method was validated using four denial-of-service EOAs. Atzei et al. [10] provided a survey on real-world attacks against Ethereum smart contracts, giving a taxonomy and discussing the vulnerabilities in detail. However, this work focused on the vulnerability assessment and did not study the attacker operations and the associated kill chain. To the best of our knowledge, our paper is the first to study cybercriminal ecosystem (e.g., attack lifecycle, attack infrastructures, campaign organization, etc.) on real-world Dapp attacks, leveraging the open and immutable transaction records kept by the Ethereum.

Security event detection and forensic. *DEFIER* investigated the problem of intrusion detection and forensic analysis, with a specific focus on Etherem Dapp attacks. Numerous studies [21, 38, 43] have looked into security event detection and forecast in various domains. Recent year witnesses the trend of understanding high-level event semantics for a more efficient and effective security event detection. Ben-Asher et

al. [12] quantitatively evaluated the effectiveness of using contextual knowledge for detecting cyber-attacks. Ma et al. [31] proposed a semantics aware program annotation to partition execution based on the application specific high level task structures. Shen et al. [44] used temporal word embedding to cluster security events under similar context and track their evolution. Hassan et al. [26] proposed a threat alert triage system that features historical and contextual information to automatically triage alerts. The closest work to our study is HOLMES [35], a real-time APT detection system that generates a high-level graph, that summarizes the attacker's kill chain steps, to identify behavior associated with *known attacks* based on frequency analysis. In contrast to previous works, the kill chain and the associated attack operations are under explored in the domain of Ethereum Dapp attacks, which turned out to be very different from the traditional APT kill chain. In our study, we first time utilize Ethereum transaction time series analysis based on graph sequence mining to learn the high-level attack operational intents, which allows us to accurately detect both known and unknown attacks.

7 Conclusion

In this paper, we report our study on Ethereum Dapp attack incidents, which consist of a sophisticated attack hierarchical structure, multiple criminal roles, and various kinds of attack behaviors. To investigate such attack incidents, we performed the *first* measurement study and forensic analysis on real-world Dapp attacks, leveraging the open and immutable transaction records kept by the Ethereum blockchain. In particular, we propose a methodology to supplement the missing attack information of Dapp incident reports. Utilizing more comprehensive attack transactions and their execution traces for each attack incident, we conduct an empirical study to recover Dapp cybercriminal's end-to-end footprints, as well as the corresponding kill chain and attack patterns. Moving forward, we believe that there is a great potential to utilize such threat intelligence to automatically investigate Dapp on a large scale. Running on 2,350,779 transactions from 104 Ethereum on-chain Dapp, our Dapp investigation tool *DEFIER*, which captures high-level attack intents, successfully identified 476,342 exploit transactions on 85 victim Dapps, which have never been reported before. It sheds on light that our understanding of Ethereum Dapp cybercrime will help more effectively defend against this emerging threat.

Acknowledgments

We wish to acknowledge the efforts of the anonymous reviewers for their insightful comments and suggestions to improve the quality of our manuscript. We also thank Boxify to share invaluable Ethereum transaction datasets with us. This work was supported in part by the NSF CNS-1618493, 1801432, 1838083 and 1850725. CAS authors was supported in part by the Key Laboratory of Network Assessment Technology of Chinese Academy of Sciences and Beijing Key Laboratory of Network Security and Protection Technology. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of our funders.

Availability

The annotated data and the implementation of *DEFIER* is available at https://drive.google.com/drive/folders/1cdD1gHNbWIS228QXmeUReougSL_k1kvf?usp=sharing.

References

[1] dapp ranking. https://www.dapp.com/ranking. Accessed Jul 2, 2019.

[2] Etherscan. https://etherscan.io/charts#generalInfo. Accessed Jul 1, 2019.

[3] get transactions by blocknumber etherscan. http://api.etherscan.io/api?module=account&action=txlist&address=YourAddress&startblock=0&endblock=99999999&sort=asc&apikey=YourApiKeyToken. Accessed: Jul 2, 2019.

[4] get transactions by pagenum etherscan. http://api.etherscan.io/api?module=account&action=txlistinternal&address=0x2c1ba59d6f58433fb1eaee7d20b26ed83bda51a3&startblock=0&endblock=2702578&sort=asc&apikey=YourApiKeyToken. Accessed: Jul 2, 2019.

[5] Gmatch4py: a graph matching library for python. https://github.com/Jacobe2169/GMatch4py.

[6] Octopus: a security analysis framework for webassembly module and blockchain smart contract. https://github.com/quoscient/octopus.

[7] Online solidity decompiler. https://ethervm.io/decompile.

[8] stateofthedapps. https://www.stateofthedapps.com. Accessed Jul 2, 2019.

[9] Charu C Aggarwal. Data mining: the textbook. Springer, 2015.

[10] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. IACR Cryptology ePrint Archive, 2016:1007, 2016.

[11] James Barton. How many ethereum smart contracts are there. https://coindiligent.com/how-many-ethereum-smart-contracts. Accessed Nov 8, 2018.

[12] Noam Ben-Asher and Cleotilde Gonzalez. Effects of cyber security knowledge on attack detection. Computers in Human Behavior, 48:51–61, 2015.

[13] Bloxy. bloxy. https://bloxy.info/. Accessed Jul 1, 2019.

[14] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Proceedings of COMPSTAT'2010, pages 177–186. Springer, 2010.

[15] Ting Chen, Yuxiao Zhu, Zihao Li, Jiachi Chen, Xiaoqi Li, Xiapu Luo, Xiaodong Lin, and Xiaosong Zhange. Understanding ethereum via graph analysis. In IEEE INFOCOM 2018-IEEE Conference on Computer Communications, pages 1484–1492. IEEE, 2018.

[16] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In Proceedings of the 2018 World Wide Web Conference on World Wide Web, pages 1409–1418. International World Wide Web Conferences Steering Committee, 2018.

[17] Chris Chinchilla. Ethereum white paper. https://github.com/ethereum/wiki/wiki/White-Paper. Accessed Jun 19, 2019.

[18] Usman W Chohan. The cryptocurrency tumblers: Risks, legality and oversight. 2017.

[19] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In International conference on machine learning, pages 2702–2711, 2016.

[20] Thibault de Balthasar and Julio Hernandez-Castro. An analysis of bitcoin laundry services. In Nordic Conference on Secure IT Systems, pages 297–312. Springer, 2017.

[21] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 1285–1298. ACM, 2017.

[22] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In Kdd, volume 96, pages 226–231, 1996.

[23] Barney G Glaser and Anselm L Strauss. Discovery of grounded theory: Strategies for qualitative research. Routledge, 2017.

[24] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. Neural Networks, 18(5-6):602–610, 2005.

[25] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining, pages 855–864. ACM, 2016.

[26] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In Network and Distributed Systems Security Symposium, 2019.

[27] Alyssa Hertig. How ethereum works. https://www.coindesk.com/information/how-ethereum-works. Accessed Mar 30, 2017.

[28] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, 1997.

[29] Paul Jaccard. The distribution of the flora in the alpine zone. 1. New phytologist, 11(2):37–50, 1912.

[30] 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.

[31] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. {MPI}: Multiple perspective attack investigation with semantic aware execution partitioning. In 26th {USENIX} Security Symposium ({USENIX} Security 17), pages 1111–1128, 2017.

[32] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[33] Patricia Yancey Martin and Barry A Turner. Grounded theory and organizational research. The journal of applied behavioral science, 22(2):141–157, 1986.

[34] Charles D Michener and Robert R Sokal. A quantitative approach to a problem in classification. Evolution, 11(2):130–162, 1957.

[35] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. arXiv preprint arXiv:1810.01594, 2018.

[36] NCC. Decentralized application security project. https://dasp.co/. Accessed Apr 10, 2019.

[37] NCC. Reentrancy. https://dasp.co/#item-1. Accessed Apr 10, 2019.

[38] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. Detection of early-stage enterprise infection by mining large-scale log data. In 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pages 45–56. IEEE, 2015.

[39] Nathaniel Popper. A hacking of more than $50 million dashes hopes in the world of virtual currency. https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html. Accessed Juln 17, 2016.

[40] SECBIT. A comprehensive solution to bugs in fomo3d-like games. https://hackernoon.com/a-comprehensive-solution-to-bugs-in-fomo3d-like-games-ab3b054f3cc5. Accessed Apr 11, 2020.

[41] SECBIT. How the winner got fomo3d prize—a detailed explanation. https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f. Accessed May 30, 2019.

[42] Beosin (Kai Sedgwick). Someone wins $3 million jackpot in ethereum ponzi fomo3d. https://news.bitcoin.com/someone-wins-3-million-jackpot-in-ethereum-ponzi-fomo3d/. Accessed Apr 11, 2020.

[43] Yun Shen, Enrico Mariconti, Pierre Antoine Vervier, and Gianluca Stringhini. Tiresias: Predicting security events through deep learning. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 592–605. ACM, 2018.

[44] Yun Shen and Gianluca Stringhini. Attack2vec: Leveraging temporal word embeddings to understand the evolution of cyberattacks. arXiv preprint arXiv:1905.12590, 2019.

[45] Beosin (Chengdu LianAn Tech). Beware! owner access could be stolen from another 3 contracts — do not ignore simple mistakes. https://medium.com/@Beosin/beware-owner-access-could-be-stolen-from-another-3-contracts-do-not-ignore-simple-mistakes-f4ebbc80db98. Accessed Jul 31, 2019.

[46] Christof Ferreira Torres and Mathis Steichen. The art of the scam: Demystifying honeypots in ethereum smart contracts. arXiv preprint arXiv:1902.06976, 2019.

[47] ubitok.io. Post-mortem investigation. https://www.kingoftheether.com/postmortem.html. Accessed May 30, 2019.

[48] web3j. Transactions — web3j 4.1.0 documentation. https://web3j.readthedocs.io/en/latest/transactions.html. Accessed Mar 30, 2017.

[49] GAVIN WOOD. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf. Accessed Jun 13, 2019.

[50] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: reverse engineering ethereum's opaque smart contracts. In 27th {USENIX} Security Symposium ({USENIX} Security 18), pages 1371–1385, 2018.

Appendix

7.1 Data formats of three types of transactions and their associated receipts

TO	0x9795***
FROM	0x1249***
VALUE	0.65 Ether
DATA	0x
GAS PRICE	1.20002x10^{-8} Ether (12.0002 Gwei)

(a) Ether transfer

TO	(empty)
FROM	0x9795***
VALUE	0 Ether
DATA	bytecode of contract
GAS PRICE	1.2x10^{-8} Ether (12 Gwei)

(b) Contract creation

TO	0x9528***
FROM	0x9795***
VALUE	0.0577307 Ether
DATA	0xc0406226 (methodID of function run())
GAS PRICE	1x10^{-8} Ether (10 Gwei)

(c) Contract call

TRANS HASH	0x9285***
FROM	0x1249***
TO	0x9795***
GAS USED	21,000
CONTRACT ADDRESS	null
STATUS	0x1

(d) Receipt of Ether transfer

TRANS HASH	0xa4c8***
FROM	0x9795***
TO	(empty)
GAS USED	624,014
CONTRACT ADDRESS	0x6be5***
STATUS	0x1

(e) Receipt of Contract creation

TRANS HASH	0x4971***
FROM	0x9795***
TO	0x9528***
GAS USED	417,124
CONTRACT ADDRESS	null
STATUS	0x1

(f) Receipt of Contract call

Figure 11: Three types of transactions supported on Ethereum.

7.2 Parameter and model selection for transaction clustering

As mentioned in Section 3.1, the parameter α and β indicate the importance of structure similarity and timing closeness when measuring TG distance

Table 13: Performance comparison under different cluster model

Method	accuracy	recall	time cost	parameters setting
k-Means [32]	0.95	0.83	84.93s	iteration number is 3; *k* is all the first transaction in sequences split by a 10-hour time window
Agglomerative Hierarchecal [34]	0.83	0.97	2h30min	*k* is all the first transaction in sequences split by a 10-hour time window
DBSCAN [22]	0.89	0.76	2h27min	eps is 0.5; the minimal points of a cluster is 2

Table 14: List of Dapp incidents reports.

Source	Report URL	Victim Dapp
PeckShield	https://blog.peckshield.com/2018/04/22/batchOverflow/	BeautyChain(BEC)
PeckShield	https://blog.peckshield.com/2018/04/25/proxyOverflow/	MESH, UGToken(UGT), SmartMesh(SMT), SmartMesh Token(SMART), MTC, First(FST), GG Token, CNY Token(CNYt)
PeckShield	https://blog.peckshield.com/2018/05/10/multiOverflow/	Social Chain (SCA)
PeckShield	https://blog.peckshield.com/2018/08/18/replay/	SmartMesh(SMT), UGToken(UGT), First(FST), MTC
PeckShield	https://blog.peckshield.com/2018/08/14/unsafemath/	MovieCredits (EMVC)
Medium	https://medium.com/coinmonks/an-inspection-on-ammbr-amr-bug-a53b4050d52	Ammbr(AMR)
4Hou	https://4hou.win/wordpress/?p=21704	Ammbr(AMR), Beauty Coin (BEAUTY), Rocket Coin (XRC), Social Chain (SCA)
BCSEC	https://bcsec.org/index/detail?id=157&tag=1	Morph
Aeternity	https://blog.aeternity.com/parity-multisig-wallet-hack-47cc507d964d	Parity
BitcoinTalk	https://bitcointalk.org/index.php?topic=1400536.60	Rubixi
Github	https://github.com/ether-camp/virtual-accelerator/issues/8	HackerGold(HKG)
Reddit	https://www.reddit.com/r/ethdev/comments/7x5rwr/tricked_by_a_honeypot_contract_or_beaten_by/	PrivateBank
Reddit	https://www.reddit.com/r/ethereum/comments/916xni/how_to_pwn_fomo3d_a_beginners_guide	Fomo3D
PeckShield	https://blog.peckshield.com/2018/07/24/fomo3d/	Fomo3D, RatScam
Medium	https://medium.com/@AnChain.AI/largest-smart-contract-attacks-in-blockchain-history-exposed-part-1-93b975a374d0	Fomo3D, LastWinner, RatScam, FomoGame
Medium	https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f	Fomo3D
Medium	https://medium.com/@Beosin/there-is-only-one-truth-god-game-attack-analysis-ea4821d27cc3	GodGame
360	http://blogs.360.cn/post/Fairness_Analysis_of_Dice2win_EN.html	Dice2Win
King of the Ether Throne	https://www.kingoftheether.com/postmortem.html	King of the Ether Throne
Reddit	https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/	GovernMental
Medium	https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe	SpankChain

(Definition 1). In our implementation, we analyzed the impact of various α and β as shown in Table 15 on the ground-truth set, and chose the combination of α and β (i.e., $\alpha = 0.9, \beta = 0.1$) with the best performance.

Also, we compared the effectiveness of different clustering algorithms, i.e., *k*-Means, DBSCAN, Agglomerative Hierarchical, on our task. The results, with pre-parameters required by cluster models, are shown in Table 13. We observe that the clustering algorithm *k*-Means outperforms other clustering algorithms in terms of accuracy and efficiency. In our study, we weight the correctness of the results and use k-Means for transaction clustering.

Table 15: Performance comparison under different distance weight

α	β	precision	recall
0.01	0.99	0.97	0.78
0.1	0.9	0.97	0.76
0.3	0.7	0.97	0.72
0.5	0.5	0.97	0.62
0.7	0.3	0.95	0.81
0.9	0.1	0.95	0.83
0.99	0.01	0.96	0.82

7.3 Data annotation

We manually examined transaction clusters to identify the adversary's intent and annotate their attack stage. Serving this purpose is the grounded theory [33], a systematic methodology that constructs a concept through methodical gathering and analysis of data in social science. More specifically, we analyzed transaction clusters through the following three stages: *coding* that identifies the anchors (e.g., multiple contract creations and self-destruction traces in a transaction, using the same contract to call several Dapps, achieving significant large profit in one transaction, etc.) that enable the key points of the annotation; *code collection and iteration* that iteratively groups anchors and aligns them to the adversary's operational intents through comparison [23] (e.g., a transaction cluster shows the operational intent of the attack propagation, if their execution traces consist of multiple contract creations and self-destruction when calling several different Dapps); *Attack stage annotation* that annotate transaction clusters' attack stage based on adversary's operational intents. Throughout the analysis, annotators intensively discussed with each other to ensure that all transaction clusters were correctly understood and evaluated. In total, it took 5 human labors around two weeks for data annotation.