Uncover the Premeditated Attacks: Detecting Exploitable Reentrancy Vulnerabilities by Identifying Attacker Contracts

Shuo Yang
Sun Yat-sen University
Zhuhai, China
yangsh233@mail2.sysu.edu.cn

Jiachi Chen*
Sun Yat-sen University
Zhuhai, China
chenjch86@mail.sysu.edu.cn

Mingyuan Huang
Sun Yat-sen University
Zhuhai, China
huangmy83@mail2.sysu.edu.cn

Zibin Zheng
Sun Yat-sen University
Zhuhai, China
zhzibin@mail.sysu.edu.cn

Yuan Huang
Sun Yat-sen University
Zhuhai, China
huangyuan5@mail.sysu.edu.cn

ABSTRACT

Reentrancy, a notorious vulnerability in smart contracts, has led to millions of dollars in financial loss. However, current smart contract vulnerability detection tools suffer from a high false positive rate in identifying contracts with reentrancy vulnerabilities. Moreover, only a small portion of the detected reentrant contracts can actually be exploited by hackers, making these tools less effective in securing the Ethereum ecosystem in practice.

In this paper, we propose BlockWatchdog, a tool that focuses on detecting reentrancy vulnerabilities by identifying attacker contracts. These attacker contracts are deployed by hackers to exploit vulnerable contracts automatically. By focusing on attacker contracts, BlockWatchdog effectively detects truly exploitable reentrancy vulnerabilities by identifying reentrant call flow. Additionally, BlockWatchdog is capable of detecting new types of reentrancy vulnerabilities caused by poor designs when using ERC tokens or user-defined interfaces, which cannot be detected by current rule-based tools. We implement BlockWatchdog using cross-contract static dataflow techniques based on attack logic obtained from an empirical study that analyzes attacker contracts from 281 attack incidents. BlockWatchdog is evaluated on 421,889 Ethereum contract bytecodes and identifies 113 attacker contracts that target 159 victim contracts, leading to the theft of Ether and tokens valued at approximately 908.6 million USD. Notably, only 18 of the identified 159 victim contracts can be reported by current reentrancy detection tools.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

---

*corresponding author

KEYWORDS

Smart Contract; Dataflow Analysis; Reentrancy; Attacker Identification; Ethereum

1 INTRODUCTION

In recent years, Ethereum has experienced significant growth in popularity and market cap [15], primarily due to its ability to support a wide range of decentralized applications (DApps) [55], such as decentralized finance (DeFi) [53] and non-fungible tokens (NFTs) [57]. This is made possible through the use of smart contracts [59], which are Turing-complete programs that run on the blockchain. However, as the value of Ethereum continues to rise, attackers are increasingly targeting contracts with vulnerabilities that can be exploited to make unfair profits. Reentrancy is one of the infamous vulnerabilities, which has caused huge financial losses [60] since the 150 million USD DAO attack in 2016 [25], and new reentrancy attacks keep popping up in more and more complex forms. For example, an attacker exploited a reentrancy vulnerability to drain approximately 1,300 ETH (1.43 million USD) from the NFT money market platform called Omni [1] by using the hook function *onERC721Received()* declared in the ERC721 standard [18].

Numerous studies have focused on detecting vulnerabilities in smart contracts [16, 23, 28], proposing various methods such as static analysis and dynamic testing to identify potential issues. However, these works face two main limitations. First, there is a high false-positive rate in detecting reentrancy vulnerabilities [60], as they cannot correctly detect some protection patterns. Furthermore, these methods mainly focus on reentrancy caused by *call.value()* operations, which cannot cover more complex reentrancy vulnerabilities (leading to false negatives) caused by poor designs when using standard ERC tokens, e.g., ERC721, or user-defined interfaces (see Section 2.2). Second, only 2.68% of contracts with reentrancy vulnerabilities can truly be exploited by hackers, and only 0.4% of the Ethers at stake could be exploited [35]. Real-world attackers

intend to evaluate the cost and benefit of an attack, but most contracts do not have the balance that can be extracted. Specifically, on Ethereum, only 3% of the contracts have a non-zero balance. Thus, most of the vulnerable contracts labeled by the tools cannot be exploited and are false alarms, which makes it less effective in securing the Ethereum ecosystem in practice.

Exploiting reentrancy vulnerabilities requires deploying malicious contracts that initiate callbacks to the victim contract. In this paper, we shift our focus from vulnerability detection to analyzing attacker contracts. To investigate how attackers implement callback logic on victims, we conduct an empirical study by analyzing 281 attack incident reports from various platforms, e.g., Twitter [49], Medium [2], and Peckshield [34], spanning from June 2016 to July 2022. These platforms provide comprehensive and timely descriptions of attack incidents, which are also adopted by other works [57]. Consequently, we summarize three types of reentrancy attack types based on the functions that attacker contracts used to make callbacks (see Section 3). Furthermore, we propose BlockWatchdog, a tool that utilizes cross-contract static dataflow techniques to identify reentrancy attacker contracts. *First*, BlockWatchdog decompiles the contract's bytecode to the intermediate representation (IR) and extracts flow and external call information in the functions. *Second*, BlockWatchdog identifies the contracts in the call chain and constructs the cross-contract control flow graph (xCFG) and the cross-contract call graph (xCG) of the contract based on dataflow rules. *Then*, it traces all call chains to perform a taint analysis to determine whether the attacker can manipulate the call chain, making itself called again to implement reentrancy. Based on detection patterns designed in collaboration with external call and dataflow information, BlockWatchdog reports whether the contract is an attacker contract or not and identifies vulnerable victim contracts.

In the experiment, we first evaluate BlockWatchdog on our collected ground-truth dataset, which contains 18 reentrancy attacker contracts. Then, we run BlockWatchdog on a real-world dataset containing 421,889 contracts' bytecodes obtained by replaying transactions from block number 10 million to 15.5 million on the Ethereum mainnet. The average detection time of it is 17.66 seconds. Furthermore, BlockWatchdog identifies 249 attacker smart contracts in this dataset, and 113 of them are labeled as true positives. Among them, 40 are 0-day attacker contracts, which involve 159 victim contracts. Ethers and tokens worth approximately 908.6 million USD in these contracts have been stolen by attackers. Furthermore, we run seven tools for reentrancy vulnerability detection on identified victim contracts; only 18 (11.3%) of them can be correctly reported.

The main contributions of our work are as follows.

- We shift the detection focus from vulnerable contracts to attacker contracts, which alleviates the high false positive problem and limited capability of current tools in finding reentrancy.
- We summarize three types of reentrancy attacks from an empirical study and introduce BlockWatchdog, a cross-contract static dataflow analysis tool to find attacker contracts and vulnerable victim contracts they target. Additionally, Block-Watchdog is extensible for users to program more rules to cover new attack types.

- We evaluate the performance of BlockWatchdog on a dataset consisting of 421,889 contracts bytecode. Our experiments show that BlockWatchdog identifies 113 attacker contracts and 159 victim contracts, which hold Ethers and tokens worth approximately 908.6 million USD. Only 18 of the 159 victims can be detected by the current tools. We publicize the source code of BlockWatchdog and the experimental results in our repository[1].

2 BACKGROUND AND MOTIVATION

2.1 Solidity Smart Contracts

A smart contract is a self-executing agreement that is enforced by the rules encoded in its code [46]. Solidity is the most popular programming language for smart contracts on Ethereum. The bytecode and transactions of the deployed smart contracts are permanently stored on the blockchain [59]. The immutability of smart contracts ensures that their code and behavior cannot be modified once deployed, and they execute automatically based on their predefined logic. Ethereum Virtual Machine (EVM) is a stack-based virtual machine that executes transactions by splitting the EVM bytecode into operation codes (opcodes) and following their instructions.

2.2 Reentrancy

The reentrancy vulnerability has resulted in significant financial losses over the past few years. There are many works that focus on detecting reentrancies caused by *call.value()* pattern [60]. Solidity smart contracts have a unique mechanism that requires any contract that receives Ethers to implement a fallback function. The fallback function will be executed if the contract receives Ether from other addresses. If the victim contract transfers Ethers to the malicious attacker contract, the malicious one can take over the control flow and repetitively call the victim in its fallback function. Many attackers have exploited this fallback mechanism to drain funds from victims. Not only those caused by *call.value()*, there are some new reentrancy types. For example, Lenf.me [24] and Omni [1] were attacked by the bad design of using ERC777 [14] and ERC721 [18] tokens, respectively.

In addition, poor design when using user-defined interfaces can also lead to reentrancy issues. Figure 1 shows the attacker contract that hacked 8.2 million USD through a reentrancy attack on IVisor [19], a liquidity management protocol of Uniswap V3 [51]. The function *delegatedTransferERC20()* (L25) is defined by the developers, which is not declared in the ERC token standard. The attacker contract injects external calls into the function *delegatedTransferERC20()* (L13-L17) invoked by the victim contract *RewardHypervisor* (L20). In detail, the attacker contract invokes the function *deposit()* (L8) of the contract *RewardHypervisor*. Then, *RewardHypervisor* calls the *delegatedTransferERC20()* (L25) of contract *from*, which is set by the attacker contract with *address(this)*, i.e., the attacker contract itself (L25). However, the attacker makes a callback to *RewardHypervisor* again to deposit again on line 8, which makes it suffer from a reentrancy vulnerability. The contract *RewardsHypervisor* does not contain *call.value()* reentrancy vulnerability type, which current detection tools focus on. Yet, it was still attacked by

---

the malicious attacker contract to make unfair gains due to the bad design when using user-defined interface *delegatedTransferERC20()*.

```
1  // the decompiled IR of the attacker contract bytecode
2  contract Attacker {
3    function 0x4a0b0c38() public payable {
4      0x28e();}
5
6    function 0x28e() private {
7      require(_pool.code.size);
8      v0, v1 = _pool.deposit(0x52b7d2dcc80cd2e4000000,
            address(this), _admin);
9      require(v0);
10     require(RETURNDATASIZE() >= 32);
11     return ;}
12
13   function delegatedTransferERC20(address varg0, address
            varg1, uint256 varg2) public payable {
14     require(msg.data.length - 4 >= 96);
15     _count += 1;
16     if (_count < 2) {
17       0x28e();}}
18 }
19 // the source code of the victim contract
20 contract RewardsHypervisor {
21   function deposit(uint256 visrDeposit, address payable
            from, address to) external returns (uint256
            shares) {
22     ...
23   if(isContract(from)) {
24     require(IVisor(from).owner() == msg.sender);
25     IVisor(from).delegatedTransferERC20(address(visr),
            address(this), visrDeposit);}
26   else {
27     visr.safeTransferFrom(from, address(this),
            visrDeposit);}
28   vvisr.mint(to, shares);}
29 }
```

Figure 1: Reentrancy attack toward Visor Finance.

2.3 Prior Research and Their Limitations

Unexploitable for the detected contracts. Previous research has focused on detecting reentrancy vulnerabilities, but most of the contracts detected are either toy contracts with no value or cannot be exploited by attackers [35]. Specifically, only 1.98% of the 23,327 reported vulnerable contracts from six academic projects have been exploited since deployment, affecting only 0.27% of the funds in the contracts [16, 21, 23, 28, 32, 48]. The reason is that the majority of Ether or tokens are held by only a small number of vulnerable contracts, which are lucrative for hackers, making most of the detected contracts unexploitable. However, detecting attacker contracts can help identify truly exploitable yet vulnerable contracts.

Poor performance in detecting reentrancy. Existing reentrancy detection tools have an extremely high false positive rate. More than 99.8% of the reentrant contracts detected by the tools [7, 28, 29, 39, 48] are false positives [60], as these tools do not detect some protection patterns, such as the reentrancy lock. To reduce false positives, detecting attacker contracts can help find those hackers who aim at truly exploitable contracts without reentrancy protections.

Furthermore, existing tools cannot cover new types of reentrancies caused by poor design when using ERC tokens, which can result in false negatives. This is because rule-based tools have limited scalability, as each rule can only check a specific reentrancy type (e.g., reentrancy caused by *call.value()*) and cannot cover emerging types, such as new ERC standards [33] (e.g., reentrancy caused by ERC721 and ERC1155 [13]), or user-defined interfaces, e.g., *delegateTransferERC20()* in Figure 1.

To address these limitations, a more general detection method is needed to reduce both false positives and false negatives. Reentrancy vulnerabilities are mainly characterized by the mutual invocation of the victim contract and the attacker contract with callback flow features, but victim contracts contain limited and non-homogeneous information, making it challenging to summarize a generic signature for rule-based tools. It motivates us to recover reentrancy features from the attacker contracts perspective.

3 ATTACKER SMART CONTRACTS FOR REENTRANCY

In this section, we present an empirical study aimed at identifying the characteristics of attacker contracts involved in historical reentrancy attacks. Our goal is to uncover reentrancy vulnerabilities that existing tools have missed. We approach the analysis of attacker contracts through a data collection process, followed by a data analysis and feature identification process, as illustrated in Figure 2 and described in subsequent subsections.
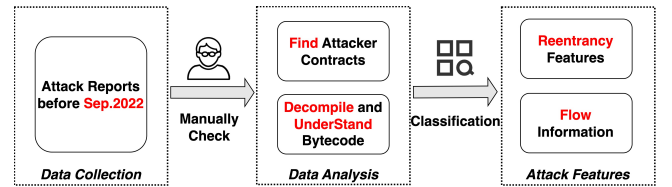

Figure 2: Workflow of finding new reentrancy types.

3.1 Data Collection

To gain a comprehensive understanding of attacks, it is necessary to collect and analyze relevant data. We began by searching for attack reports published by reputable blockchain security firms such as PeckShield, as well as information from social media platforms like Twitter and publishing platforms like Medium. In total, we identified 281 incidents that occurred between June 2016 and July 2022. For each report, we recorded key information such as the target project, the victim contract address, the attack time, and the associated losses. All the data collected and their source links are available for public access in our open repository.

3.2 Data Analysis

Attacker contract identification. As we do not know how hackers perform the attack, we intend to find the attacker contracts to analyze the attack logic from the collected data. Specifically, two of our authors, both with more than two years of experience in blockchain security, manually analyzed the 281 incidents collected using the open card sorting approach [43]. During the manual

check, we identified two distinct types of attacks: those directly attacked by externally owned accounts (EOAs) of hackers and those attacked by attacker contracts that hackers deployed.

Table 1: Attack Types Obtained from Collected Data

	DoS	BR	IO	RE	IA	CI	CAD	FL	Others
AC/EOA	◑	●	○	●	○	○	○	●	◑
CA	☹	☹	☹	☹	☹	☹	☹	☹	☹

We then classify the attack types based on the necessity of an attacker contract and the availability of the attacker contract's source code. Table 1 shows the eight types of attack that we identified from our 281 collected incidents. We use the "Others" category to cover attacks that target specific design flaws of victims, e.g., infinite approval to vulnerable contracts [37]. The ● and ○ symbols in the column "Attacker Contract" represent attack types that require deploying the attacker contract (AC) or using EOA transactions, respectively. The symbol ◑ represents attack types that do not require the deployment of an attacker contract in some cases. The ☹ symbol in the "Code Availability" (CA) column represents attacker contracts whose source code is not available. Among the 281 samples, we found 31 attacker contracts from 28 reports, classified as Denial of Service (DoS), Bad Randomness (BR), Reentrancy (RE), Flashloan (FL), and Others. Attacks such as Integer Overflow (IO), Improper Authentication (IA), Call Injection (CI), and Call-after-destruct (CAD), which do not involve attacker contracts, are out of the scope of our analysis. As this paper focuses on new types of reentrancy vulnerabilities missed by existing vulnerability detection tools, we will illustrate reentrancy vulnerabilities from the perspective of the 18 reentrancy attacker contracts collected.
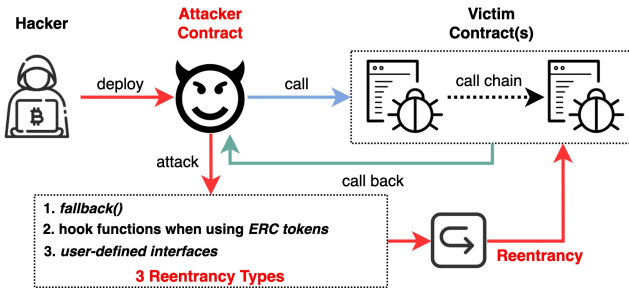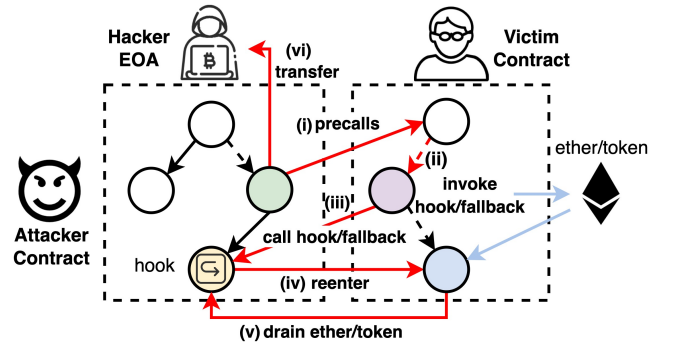

Figure 3: An overview of reentrancy attacks.

Decompilation and understanding. To gain a deeper understanding of the attacker behaviors employed by reentrancy attacker contracts, we collected the bytecode of attacker contracts for analysis, as none of the identified attacker contracts released their source code to the public. We decompiled the EVM bytecode to recover a readable intermediate representation (IR) of the attacker contract. We then followed the attack process and description according to the report to understand how the attacker implements the attack logic from its bytecode. Figure 3 provides an overview of how attacker contracts exploit victim contracts. The hacker first programs the attacker contract and deploys it on the blockchain. Subsequently,

the attacker contract automatically executes and initiates external calls to the victim contracts. Notably, in reentrancy attacks, the attacker contract can pass parameters to victim contracts, which makes them call back to the attacker contract again, and there can be multiple victims in this call chain. As shown in Figure 3, we summarize three types of reentrancy based on the functions utilized by attacker contracts to perform reentrancy, i.e., (1) *fallback()*, poor designs when using (2) *ERC tokens*, or (3) *user-defined interfaces*. Specifically, the attacker contract can implement reentrancy logic in the *fallback()* function when receiving Ethers. It can also inject callbacks into hook functions when using ERC tokens, e.g., hook function *onERC721Received* when using ERC721, or user-defined interfaces, e.g., the case shown in Figure 1, to implement reentrancy.

3.3 Attacker Contract Features

Figure 4 shows an example that illustrates the high-level features of the reentrancy attack focusing on the call flow. To perform the reentrancy attack, the attacker contract (1) first calls the victim contract (step i in Figure 4) to make the victim invoke a callback (step ii) to the attacker's hook function or transfer Ether to the attacker contract (step iii), (2) then the attacker contract calls the victim again in the hook function or the fallback function, and reenters (step iv) to invoke functions that can generate unfair profits (step v), (3) next, the profits can be transferred to the attacker EOA (step vi) to complete the reentrancy attack. This call flow information shows how the attacker contract interacts with the victim contract. The summarized three types of reentrancy help us identify the reentrancy from function-level call information. The design of our method shown in the following section is based on these features obtained from our empirical study.

Figure 4: An example to illustrate reentrancy attack.

4 METHODOLOGY

In this section, we introduce the BlockWatchdog tool, which can detect attacker contracts that perform reentrancy attacks. We first give an overview of the approach and then provide the details from the perspectives of flow information extraction, cross-contract static analysis, and attacker contract detection.

4.1 Overview

BlockWatchdog consists of three main components: *Decompiler*, *Cross-Contract Dataflow Analyzer*, and *Attack Identifier*. Figure 5

shows an overview of the BlockWatchdog approach. The tool can accept a contract bytecode or a real address on Ethereum as input. If a contract address is provided, the tool retrieves the bytecode from the Web3 API [52]. BlockWatchdog decompiles the bytecode to the IR and extracts critical flow information for dataflow analysis in *Decompiler*. Next, the *Cross-contract Dataflow Analyzer* constructs the xCFG and xCG of the contract according to the information obtained from the decompilation. We use xCFG and xCG to bridge the flow information of all contracts in the call chains for taint analysis. Taints originating from the attacker contract are propagated through the function call arguments and returns based on designed transfer rules while tracing all possible call chains in the xCG. Finally, the *Attack Identifier* identifies and reports the three attack types based on our detection patterns based on the result of the taint flow analysis.

To implement BlockWatchdog, we adopt a public node provided by Alchemy [3] to request data from the blockchain. Specifically, we use the Web3 API *getCode* to fetch the bytecode of a contract and use *getStorageAt* to obtain the storage data in a specific slot and offset from a contract account. For decompilation, we use the EVM bytecode decompiler Elipmoc, which improves over all the notable past decompilers [17, 22]. Elipmoc can disassemble the EVM bytecode into EVM opcodes and construct the control flow graph based on identifying flow-related opcodes like JUMP and JUMPI. The function borders and the IR are then recovered for further analysis.
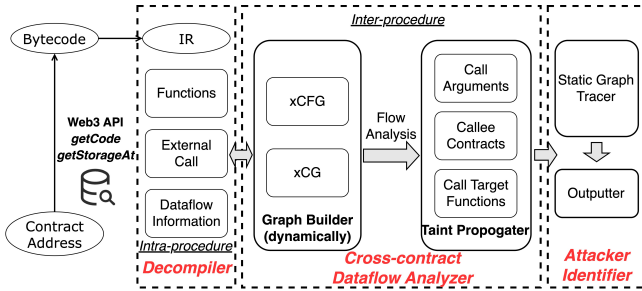

Figure 5: An overview of the approach of BlockWatchdog.

4.2 Flow Information Extraction

When analyzing an input contract, the first step in BlockWatchdog is to decompile its EVM bytecode to the IR. In the *Decompiler* component, we extract flow information from the IR to enable dataflow analysis and attacker vector identification.

Constant contract address and function signature identification. To conduct cross-contract static dataflow analysis, we need to identify the target contracts and functions that the input contract intends to call. Based on our analysis of previous attack incidents, we observe that attacker contracts typically hard-code the contract address with which they want to interact or store the target contract's address in storage. Therefore, we use the decompiler to identify the constant value (conforming to EIP-55 [11]) of the callee contract address. The slot number with specific offsets to locate the storage address can be obtained through the Web3 API *getStorageAt*. Additionally, we identify the function signatures through

decompilation, which specifies which functions can be invoked in each external call. For each external call identified in the caller contract, we obtain the callee contract and function signature to locate the callee site of the contract being called, allowing us to construct the entire xCFG and xCG for inter-procedure cross-contract analysis.

Flow and external call information extraction. To extract the flow information from a contract, we first obtain every entrance and exit of the contract. For every public function identified by the decompiler, we extract its function arguments and return parameters, which denote the start and end points of the dataflow within the function, respectively. Function arguments can also flow to the parameters of external calls made within the function, while the return values of external calls can flow back to the function's return parameters. Therefore, we design five dataflow rules in Table 2 for each public function in the contract. We illustrate each rule based on Figure 6. Taking the *FuncArgToCallArg* and *FuncArgToCallee* as an example, in Figure 6, function argument *v2* of the function *bar()* (L2-L10) flows to the arguments of the external call *target.foo()* (L4) by rule *FuncArgToCallArg*. In contract *target*, the function argument *v1* of *foo()* (L16-L22), which is set by the caller contract *from* (line (i) in Figure 6), can flow to the callee variable *v1* of the call operation *v1.hook(v2)* (L19) through rule *FuncArgToCallee* (line (ii) in Figure 6). By combining these two rules, we can obtain the flow information that *address(this)* (L4) set by contract *from* can flow to the callee variable *v1* (L19) in contract *target*. In this case, the contract *from* can make the contract *target* call its implemented function *hook* through the dataflow process. Then, the call flow returns to the contract *from*, which makes it capable of calling back to the *target.foo()* again (L12), and leads to the reentrancy.

Table 2: Dataflow Rules in Intra-Procedure Analysis

Flow Type	Meaning
FuncArgToCallArg	flow from function arguments to call arguments
FuncArgToFuncRet	flow from function arguments to function returns
FuncArgToCallee	flow from function arguments to callee variables
CallRetToCallArg	flow from call returns to call arguments
CallRetToFuncRet	flow from call returns to function returns

Additionally, we focus on whether the input contract implements an external call in the hook or fallback function, which attackers can leverage to make other contracts call back and succeed in reentering the attacker contract. We summarize six hook functions from five Ethereum Improvement Proposals (EIPs) [10] that we find to be involved in our collected reentrancy attacker contracts, as shown in Table 3. Noticeably, we list these hook functions to illustrate the features of the attacker contracts; our BlockWatchdog does not rely on these specific function signatures but focuses on the call flow features. Furthermore, the fallback function called when receiving Ethers and user-defined interfaces, e.g., *IVisor.delegatedTransferERC20()*, can also be used to perform reentrancy. Therefore, based on the external call information found in

```
1  contract from {
2  function bar(v1, v2) external {   😈
3    // FuncArgToCallArg(v2)
4    ret = target.foo(address(this), v2);
5    // CallRetToCallArg(ret)
6    addr.ret2Arg(ret);
7    add_v1 = v1 + 1;
8    // FuncArgToFuncRet(v1)
9    // CallRetToFuncRet(ret)
10   return add_v1, ret;}
11 function hook(v2) external {
12   target.foo(address(this), v2);}
13 }          Reentrancy!
14 (iii)        (iv)
15 contract target {
16 function foo(v1, v2) external {
17   // FuncArgToCallee(v1)
18 (ii)  // FuncArgToCallArg(v2)
19   v1.hook(v2);
20   ret = v2 + 1;
21   // FuncArgToFuncRet(ret)
22   return ret;}
23 }
```

target
&
(i) address(this)

can **_manipulated_**
by attacker
contract
to call victim
premeditatedly

Figure 6: Toy contracts for flow information illustration.

these types of functions and flow information extracted during the intra-procedure analysis, BlockWatchdog determines whether a reentrancy attacker contract is present.

Table 3: Hook Functions Declared by EIP

Standard	Function Name	Function Signature
EIP-20	*transferFrom*	0x01c6adc3
EIP-721	*onERC721Received*	0x150b7a02
EIP-1155	*onERC1155Received*	0xf23a6e61
EIP-777	*tokensToSend*	0x75ab9782
EIP-777	*tokensReceived*	0x0023de29
EIP-1820	*canImplementInterfaceForAddress*	0x249cb3fa

4.3 Cross-contract Static Analysis

In this subsection, we describe how we use decompiled intermediate representation (IR) and intra-procedure information to construct the xCFG and xCG of the attacker contract and its interacting contracts. Algorithm 1 presents an overview of how to identify the attacker contract. We first construct the call chain of every public function that contains an external call E_f using the extracted flow information (L1). Then, we use the depth-first search (DFS) algorithm for each public function f to find its external call target contracts and functions to construct the xCFG and xCG (L2-L3). We apply the tainted source identification rules to find the tainted source s in the input contract's function (L4). Then, we use sink identification rules for each call chain in the constructed xCFG to find sensitive variables t that can cause the attack (L6), such as function arguments flowing to callee variables. To determine whether contract C can successfully make the other contract call back to itself, we use the transfer rules from source to sink to obtain reachability (L7) and find possible attack call chains.

4.3.1 xCFG Construction & Call Chain Gathering. To perform static dataflow analysis, we first need to construct the xCG and xCFG

Algorithm 1: Cross-contract Static Analysis for Identifying Attacks

---

input: C, the input contract bytecode
output: $AC \leftarrow []$, the list of possible attack call chains
1: $E_f \leftarrow findFunctionsWithExternalCall(C)$
2: **for each** function $f \in E_f$ **do**
3:     $P_c \leftarrow searchCallPathsByDFS(f)$
4:     $s \leftarrow getSource(f)$
5:     **for each** path $p \in P_c$ **do**
6:         $t \leftarrow getPossibleSink(p)$
7:         **if** $isReachable(s, t, rules)$
8:             $AC \leftarrow AC \cup p, break$
9: **return** AC

---

of the input contract. We obtain the bytecode of the interacted contracts based on the contract address identified during decompilation. Then, for every external call of the input contract, we find the call-target contract address and function signature. Based on the tuple <*callsite*, *caller_address*, *caller_funcSign*, *target_contract*, *target_funcSign*>, we find every call site that executes the *CALL* opcode and its call target. We then use the DFS algorithm to construct the xCFG and gather all possible call chains originating from the public functions of the input contract to construct xCG for the dataflow analysis.

4.3.2 Cross-contract Dataflow Analysis. We define the tainted source, sink site, and transfer rules in our dataflow analysis for identifying attacks in Table 4. For a contract C and its external call f with arguments set A in the example, we define all external call arguments A_s in contract C as tainted. Then, for every contract called, we mark every callee variable that determines the call target address C_t as the sink site. We apply the rules shown in Table 2 to determine whether the tainted source can flow to the sink site. If there is a possible path for that dataflow path, it is possible that the input contract can make the called contract call a specific address that the attacker designed, which helps us find whether there is reentrancy.

Table 4: Dataflow Rules for Identifying Attacker Contracts

Example	$C.f(A)$, f is an external call to contract C; A is the set of arguments
Source	external call arguments set A_s of input contract
Rules	5 intra-procedure dataflow rules in Table 2
Sink	callee C_t of external calls in called contracts

4.4 Attacker Contract Detection

In this subsection, we present specific rules for detecting reentrancy attacks in the attacker contract using flow information and cross-contract dataflow analysis.

The reentrancy attack can be reflected in the call chain that we recover by cross-contract static analysis. Specifically, we detect the attacker contract that can perform a reentrancy attack in three steps. Step 1: We first determine whether there is a call path that causes tainted variables to flow to the sink site using the rules we designed in the cross-contract analysis illustrated in Section 4.3.2. For example, in Figure 6, the attacker can call function *bar()* (L2-L10)

to invoke external call *target.foo()* (L16-L22). The function argument *v1* (L16) can flow to the callee of the external call *v1.hook(v2)* (L19) in contract *target*, which means contract *from* can manipulate the call target of *v1.hook()*. We then find the target call function *hook()* (L11-L12) of the reachable sink site *v1.hook()* (L19). Step 2: We determine whether there is an implementation of the function *hook()* (L11-L12) of the input contract, i.e., contract *from*. Step 3: If Step 2 is true, we find the call target contract address and function signature to determine whether they are visited in the call path, and judge whether there is a reentrancy attack. In the example, contract *from* calls back to the function *foo()* again in the function *hook()* to perform reentrancy.

We use the following expression to define the conditions for a reentrancy attack. We first find the reachability of the sink site in contract *tar*, which calls the function f of contract *to*. The address of contract *to* can be set by the attacker, which is the tainted source denoted by A_s. Furthermore, the function f should be implemented in the attacker contract's public function list C_F, and the external calls f_{EC}, including the call target address and function signature, should be visited in the call path. If all conditions are met, we consider it a reentrancy attack.

$$Reentrancy \Leftarrow \frac{Reachable(tar, f, to), \ to \in A_s}{f \in C_F, \ f_{EC} \in Visited}$$

5 EVALUATION

In this section, we evaluate the effectiveness of BlockWatchdog based on the ground truth dataset collected from attack reports and a large-scale dataset obtained through blockchain transaction replay.

5.1 Evaluation Setup

The experiment was conducted on a server running Ubuntu 20.04.1 LTS and equipped with 18 Intel(R) Core(TM) i9-10980XE CPUs @3.00GHz and 250 GB memory.

Dataset. We use two datasets to evaluate BlockWatchdog. *The first* one is the ground truth dataset, which comprises reentrancy attacker contracts that we collect by analyzing the attack incidents reports. This dataset contains 18 attacker contracts from 15 incidents. *The second* large-scale consists of 421,889 real-world contract bytecode (both creation and runtime). These contracts are obtained via replaying transactions from block number 10 million to 15.5 million on the Ethereum mainnet.

Evaluation Metrics. We summarize the following research questions (RQs) to evaluate BlockWatchdog.

RQ1. How effectively is BlockWatchdog in detecting reentrancy attacks in the ground-truth dataset?

RQ2. How is the performance of BlockWatchdog in finding attacker contracts in the large-scale dataset?

RQ3. How much financial loss is caused by the identified attacks, and are vulnerability detection tools able to find those vulnerable victims?

5.2 Answer to RQ1: Effectiveness on the Ground Truth Dataset

To answer RQ1, we run BlockWatchdog on our ground truth dataset with 18 attacker contracts from our collected reports. BlockWatchdog correctly reports 15 out of 18 reentrancy attacker contracts, as shown in Table 5. The second and third columns show the name of the DApp and the time it was attacked; the fourth and fifth columns represent the loss of the attack and the platform the DApp was deployed. Columns sixth to eighth show the address of the exploiter, the attacker contract, and the victim contract in the attack, respectively. The last column denotes whether BlockWatchdog can identify the attacker contract. It is noticeable that we find two exploiters (0xce and 0x80) in the Cream Finance attack from our collected incidents, and these two exploiters deployed 2 (0xbd and 0x38) and 1 (0x32) attacker contracts, respectively, to attack the same victim contract (0xce). Similarly, the exploiter (0x61) that attacked the Fei protocol and Rari DApps deployed two attacker contracts (0xE3 and 0x32) to attack the victim contract (0xfb). We find that Block-Watchdog fails to identify three reentrancy attacker contracts due to two reasons. The first factor is the inability of BlockWatchdog to recover the call chain when the call target addresses and functions in the attacker contracts are obtained from memory. Given that the memory value cannot be determined via static analysis, Block-Watchdog relies on the constant value of callee addresses or call target function signatures for complete call chain recovery. In the cases of the Fei Protocol and Rari incident, BlockWatchdog cannot identify them because the call target contract address is loaded from the memory. However, the memory value can only be determined during runtime, making BlockWatchdog fail to deduce the call target and recover the call chain. The second reason pertains to the limitations in the function signature identification of Block-Watchdog. Since BlockWatchdog is based on Elipmoc [17], which may not be able to recover all function signatures, this limitation may lead BlockWatchdog to fail to identify some external calls. For example, BlockWatchdog cannot detect the reentrancy attack in the Spankchain incident, as the call target function *LCOpenTimeout()* is not identified in the victim contract. As the dataflow procedure ends with an unknown target, BlockWatchdog fails to recover the call chain and identify the reentrancy. Despite such cases, the recall of BlockWatchdog reaches 83.3% in our experiments.

5.3 Answer to RQ2: Attacker Contracts Detection in a Large-scale Dataset

To answer RQ2, we ran BlockWatchdog on 421,889 smart contracts obtained from blockchain transaction replay. The experimental results show that BlockWatchdog reports 253 contracts as attacker contracts. We next evaluate the performance of BlockWatchdog, using a manual labeling process. Two authors manually inspect the 253 samples reported as attacker contracts, following a four-step procedure. *Firstly*, the decompiled intermediate representation (IR) of the attacker contract was thoroughly examined. *Secondly*, the function that performs the attack reported by BlockWatchdog is located. *Thirdly*, the call chains reported by BlockWatchdog are checked. *Finally*, based on these examinations, it is determined

Table 5: Attacker Contracts in the Dataset and Detection Results of BlockWatchdog (BW)

#	DApp	Attack Time	Loss ($)	Platform	Exploiter	Attacker Contract	Victim Contract	BW
1	Spankchain	2018/10/9	38K	ETH	0xcf267eA3f1eb	0xc5918a927C4F	0xf91546835f75	✗
2	Uniswap	2020/4/18	220K	ETH	0x60f3FdB85B2F	0xBD2250D713bf	0x1f9840a85d5a	✓
3	Lendf.Me	2020/4/19	24.7M	ETH	0xa9bf70a420d3	0x538359785a8D	0x0eEe3E3828A4	✓
4	Akropolis	2020/11/12	2M	ETH	0xe2307837524D	0x38c40427efbA	0x1cec0e358f88	✓
5	DeFiPie	2021/7/12	350K	BSC	0xf6f43f77ef9e	0x6d741523F1Fc	0x607C794cDa77	✓
6	xSurge	2021/8/16	25M	BSC	0x59c686272e6f	0x1514aaa4dcf5	0xE1E1Aa58983F	✓
7	Cream Finance	2021/8/31	5M	ETH	0xce1f4b4f1722	0xbd51Cb8c06F7	0xce1f4b4f1722	✓
						0x38c40427efbA		✓
					0x8036EbD0Fc9C	0x32d77947aACa		✓
8	Grim Finance	2021/10/16	30M	FTM	0xDefC385D7038	0xb08cCb39741d	0x279b2c897737	✓
9	Visor Finance	2021/12/21	8.2M	ETH	0x8efab89b497b	0x10C509AA9ab2	0xc9f27a50f825	✓
10	Paraluni	2022/3/13	1.7M	BSC	0xA386F30853A7	0x4770b5cb9d51	0x94bC1d555E63	✓
11	Agave & Hundred	2022/3/15	5.5M	ETH	0xcE1F4B4F1722	0x38c40427efbA	0xf8D1677c8a0c	✓
12	Revest	2022/3/27	120K	ETH	0xef967ece5322	0xb480ac726528	0x2320a28f5233	✓
13	Fei Protocal & Rari	2022/4/30	80.3M	ETH	0x6162759eDAd7	0xE39f3C40966D	0xfbD8Aaf46Ab3	✗
						0x32075bAd9050		✗
14	Omni	2022/7/10	1.4M	ETH	0x00000000c251	0x3c10e78343c4	0x3c10e78343c4	✓
15	SushiBar	2022/10/25	15K	ETH	0x8ca72f46056d	0x9C5A2A643152	0x2321537fd8EF	✓

whether a contract is an attacker contract, and the associated financial loss is recorded. The labeled results show that 113 samples are indeed attacker contracts that perform reentrancy attacks.

In the large-scale experiment, some metrics are collected to help us evaluate BlockWatchdog more comprehensively. For the structure of the constructed xCFG and xCG, the average number of visited contracts and the call depth of the recovered xCG are 0.95 and 0.21, respectively. The same indicators of true attacker contracts are 7.66 and 2.55, respectively. This indicates that many attacker contracts directly hardcode the victim contracts' addresses for implementing the attack. In addition, the average detection time for BlockWatchdog to analyze an attacker contract is 17.66 seconds. An attacker contract can interact with a maximum number of 105 contracts in a single call chain, with the maximum call depth reaching 21. Such deep call chains involving multiple contracts and functions expose the limitations of simple pattern-based rules in covering complex reentrancy attacks.

Table 6: Top 5 Hook Functions and Call Target Functions in Identified Attacker Contracts with Occurrence Times

Hook Function	Times	Call Target	Times
uniswapV2Call	73	*balanceOf*	111
onFlashLoan	13	*transfer*	102
onERC721Received	9	*approve*	53
delegatedTransferERC20	2	*withdraw*	25
onERC1155Received	2	*deposit*	23

Table 6 shows the signatures of the call targets in the hook function of the attacker contracts from our labeled dataset. The *uniswapV2Call()* is a required hook function in Uniswap V2 [50], while *onFlashloan()* is declared by ERC3156 [12]. We also have some interesting findings related to the design of attacker contracts. For

example, 19 attacker contracts were designed with attack logic to hack victim contracts, but these attempts failed, resulting in reverted transactions. Most of these contracts only have two transactions, i.e., the contract creation transaction and the failed attack transaction. Moreover, some exploitation function names, such as *Attack*, *Rugpull*, *Exploit*, and *Trigger*, are commonly used in attacker contracts. This interesting discovery provides information for function signature or name identification to find potential attacker contracts, and provides insights into the development preferences of hackers.

False positives. We identify two types of false positives generated by BlockWatchdog when detecting reentrancy attacker contracts. The first type involves the use of *getter* functions to make external calls in the reentrant hook function, without performing any other profitable external call operations. The second type involves the usage of a permission check mechanism, where some contracts use *msg.sender* as the transfer target or to constrain the caller of the hook function, with no intention of making external calls to attack others. BlockWatchdog reports all cases based on the reentrancy path to minimize false negatives, which may generate false positives.

5.4 Answer to RQ3: Financial Loss of Victims

During the labeling process, we found that the total financial loss caused by the true positive attacks was 908.4 million USD. This loss comprised approximately 840 Ethers (about 1.7 million USD) and tokens worth 906.9 million USD. Notably, the loss of tokens accounted for 99.8% of the total financial loss, indicating that new types of reentrancy attacks are primarily caused by poor designs when using and transferring ERC tokens, rather than Ether transfers.

Table 7 shows the top 5 attacks identified by BlockWatchdog, ranked by their financial loss. We find that vulnerable contracts may be attacked multiple times if they have been exploited successfully

once. Specifically, BlockWatchdog identified 9, 4, and 3 attacker contracts that were deployed to attack projects Cream Finance [9], Omni [1], and Visor Finance [19], respectively, resulting in a total financial loss of 906.7 million USD. Cream Finance was hacked by the hook function *tokensReceived()* in the attacker contract. It implements multiple token-borrow logic to perform the flash loan attack in a reentrancy call path, which demonstrates the complexity of the new types of reentrancy attacks. Omni was attacked by attacker contracts that implemented reentrancy logic in the hook function *onERC721Received()* when transferring NFTs. Hackers repeatedly minted, borrowed, and withdrew NFTs before changing the liquidation state. Visor Finance was exploited by attacker contracts that implemented *delegatedTransferERC20()*, which is a user-defined hook function, as shown in Figure 1. In addition, BlockWatchdog identified 40 zero-day attacker contracts and a total of 159 victim contracts, which were targeted by the attacker contracts to perform reentrancy attacks.

Table 7: Top 5 Attacks Ranked by Financial Loss

Attaker Contract	Hook Function	Loss (USD)
0x10c509aa9ab2	*delegatedTransferERC20*	904 million
0xc51bdc9aebba	*tokensReceived*	0.56 million
0x86f28c7030bd	*onERC721Received*	0.28 million
0xbc82ab5a8223	*tokensReceived*	0.28 million
0x3292818dB514	*uniswapV2Call*	0.28 million

6 DISCUSSION

In this section, we first give a case study to show how BlockWatchdog can enhance the security of Ethereum by identifying attacker contracts. We then present the capability of BlockWatchdog to find vulnerabilities and discuss the limitations of our work.

6.1 Case Study

Figure 7 shows a code snippet of the attacker contract that aimed to attack Revest [40] identified by BlockWatchdog. This attacker contract was first deployed at Mar-27-2022 01:10:05 AM +UTC, and about half an hour later, the first attack was launched at Mar-27-2022 01:41:46 AM +UTC. Specifically, the function with signature *0xdd869c35* was invoked by the exploiter to call victims. Then, the call chain was turned back and controlled by the attacker contract twice through function invocation by the hook functions, i.e., *uniswapV2Call()* (L1-L4) and *onERC1155Received()* (L6-L17). The attacker contract queried for the next NFT id (function *getNextId()* (L12)) and deposited another one via the function *depositAdditionalFNFT()* (L13). Finally, another NFT was successfully minted to the attacker without being paid in this reentrancy attack. It takes BlockWatchdog 64 seconds to identify this attacker contract and its victim contract. The time gap of half an hour between the deployment of the attacker contract and the actual attack far exceeds the time costs of the detection process. This provides an opportunity for whitehats to front-run the hackers and protect the vulnerable deployed contracts. Specifically, whitehats can copy the bytecode of the attacker contract to perform the imitation attack [38] before the real attack is launched. Therefore, the intended profits of hackers

are extracted by whitehats, thus saving possible financial losses. In addition, BlockWatchdog takes 25 seconds to detect this attacker contract (shown in Figure 1) and the potentially exploitable victim in Visor Finance [19]. Since the attack occurred about 6 minutes after the deployment of the attacker contract, the protection can be performed in this time gap.

```
1  function uniswapV2Call(address varg0, uint256 varg1,
        uint256 varg2, bytes varg3) public nonPayable {
2    v18, v19 = stor_18_0_19.call(0x2e236bc, address(this),
          1, ...);
3    ...
4    v39 = stor_18_0_19.withdrawFNFT(v36, 1 + v2[0]);}
5
6  function onERC1155Received(address varg0, address varg1,
        uint256 varg2, uint256 varg3, bytes varg4) public
        nonPayable {
7    ...
8    if (_onERC1155Received != 0) {
9      if (_onERC1155Received == 1) {
10       v0 = 0x2007(_onERC1155Received);
11       _onERC1155Received = v0;
12       v1, v2 = stor_19_0_19.getNextId();
13       v3, v4 = stor_18_0_19.depositAdditionalToFNFT(v2 -
              1, stor_4, 1);}
14     } else {
15       v5 = 0x2007(_onERC1155Received);
16       _onERC1155Received = v5;}
17    return 0xf23a6e61;}
```

Figure 7: The attacker contract that hacked the Revest.

The above case study shows the practicality of timely protection by using our BlockWatchdog. In the blockchain system, attacks are irreversible, making it critical to detect potentially threatening contracts before the start of any transaction. BlockWatchdog can quickly identify such vulnerable contracts, thereby improving the security of the Ethereum ecosystem in two ways.

Firstly, our tool can be used for the real-time detection of attacker contracts on Ethereum, allowing security firms to report suspicious attacker contracts in minutes. Since we do not require transaction information, it is possible to prevent attacks. Whitehats [54], Etherscan [15], and security analysis firms, such as Consensys [8], can quickly identify potential attacker contracts and victims with attack footprints reported by BlockWatchdog (17.66 seconds on average) before attack transactions are sent (as shown in the above case). However, it is crucial to address the ethical implications of such preemptive actions in this paper. While our approach facilitates early detection and provides an opportunity for rapid response, we consciously do not perform on-chain frontrunning that could be deemed as ethical issues or potential disruptiveness to the blockchain's integrity. Furthermore, we use transaction replay only to retrieve the bytecode of the deployed smart contracts.

Secondly, our tool can find new types of reentrancy vulnerabilities that other tools may have missed. Security firms can use BlockWatchdog to identify undiscovered attacker contracts and reentrancy vulnerabilities in practice. Platforms like Etherscan label the attacker's EOA as an attacker account, and warn about newly deployed contracts and related transactions.

For contract developers, to prevent their contracts from being attacked after the deployment, the external call target contract's address should be verified in "sensitive" functions, e.g., token transfer or swap logic, thus blocking the premeditated attacks that malicious attacker contracts could launch.

6.2 Capability of Finding Vulnerabilities

BlockWatchdog finds a significant number of exploitable victim contracts with exploitable vulnerabilities, and most of them cannot be detected by current tools. We choose seven reentrancy detection tools to detect victim contracts: Mythril [30], NFTGuard [57], Oyente [28], Sailfish [39], Securify1 [48], Securify2 [42], and Smartian [7], referring to the four select rules [57, 60], i.e., (1) availability of the tool source code; (2) usability of the command-line interface for large-scale experiment; (3) supporting Solidity source code; (4) ability to report vulnerable code location for manual examination. We run these seven tools on 159 victim contracts, and all outputs are given in our open repository.

The results show that only Mythril and Sailfish report 1 and 17 victims, respectively, while the other five tools do not report any reentrancy vulnerability. Therefore, only 18 out of 159 (11.3%) victims can be detected by current tools. As BlockWatchdog extracts more features from attacker contracts, which may be missed by other works, BlockWatchdog can more effectively identify exploitable contracts with reentrancy vulnerabilities in practice with better performance and generalizability. There are three reasons for the results. First, many identified attacker contracts extract tokens but not Ethers from victims. However, most existing tools focus on *call.value()*, which only involves Ether transfer. Second, rule-based methods struggle to cover issues caused by patterns that have not been previously reported, and it is even harder to identify reentrancies related to user-defined interfaces that cannot be generalized into a detection pattern. Third, existing tools detect reentrancy based on state modification inconsistency. However, new reentrancy attacks, *e.g.*, read-only reentrancy, can make use of just view functions, e.g., *balanceOf()* and *getPrice()* to implement reentrancy. These view functions do not modify any state and are usually not protected, making existing tools difficult to detect.

6.3 Limitations

Despite the strengths of BlockWatchdog, we identify three potential limitations. *First*, BlockWatchdog reports whether an input contract is an attacker contract or not based solely on static analysis without transaction information. We can recover the possible call chains of the input contract, including transaction data, which can help enhance the precision of detection. As we aim to deploy Block-Watchdog as a real-time detection platform for Ethereum in the future, transactions are considered as additional information to achieve a more precise detection result *Second*, we identify interacted contracts from the constant address or storage, and by default, we rely on the assumption that the attacker contract hardcodes the victims' addresses. Although we find that all the collected cases belong to these two scenarios, it is possible that the attacker contracts can set the target contract in the arguments of the function, which we cannot obtain through static analysis, thus leading to false negatives. In addition, our taint analysis does not account for

conditional checks when tracing paths, which yields false positives. *Third*, we summarize the attack types and design the detection rules based on the reported attack incidents. Therefore, it is possible that we may not cover new attack types that have not yet occurred or been reported. Overall, despite the above limitations, we focus on detecting new types of reentrancy vulnerabilities that other tools cannot cover. Other types of attacks that involve attacker contracts mentioned in Section 3 are not addressed in this paper, which we will cover in future work.

6.4 Threats to Validity

Regarding our experiment, the manual labeling process may have introduced errors in differentiating false positives and true positives. However, we have used a double-check process to mitigate this issue and updated the labeled dataset in a timely manner to ensure accuracy. We validated whether there was an attack on Etherscan using the call chains reported by BlockWatchdog, collaborated with the transaction trace recorded on the online transaction trace explorer Phalcon [36], and recorded financial loss according to the transaction information obtained from Etherscan. Another threat to validity is that we did not verify that vulnerable contracts can be exploited due to ethical concerns about attacking them. We intend to address this in our future work.

There are some other types of attacker contracts, e.g., Bad Randomness and Flashloan, as we mention in 3.2, which are not covered by this work. The design of our BlockWatchdog focuses on call flow analysis, which makes it possible to be extended to detect Flashloan attacker contracts that also contain some flow features. However, more contract semantics and operational features should be analyzed to cover other attacker contract types like Bad Randomness, which requires understanding specific contract behaviors.

Despite the limitations mentioned above, BlockWatchdog has detected 40 zero-day attacker smart contracts that were previously unreported, which had reentrancy features. Since BlockWatchdog does not require transaction information and can provide detection results within minutes, it can monitor newly deployed contracts and detect attacker contracts before they can execute attacks, preventing potential attacks and heavy financial losses.

7 RELATED WORK

Reentrancy detection tools for smart contracts. As reentrancy is one of the notorious vulnerabilities in smart contracts [41], many program analysis tools have been developed to detect such issues by static analysis or dynamic testing [4–6]. The goal of these tools is to prevent vulnerable contracts from being deployed on the blockchain. For example, Oyente [28], Securify [48], Mythril [30], and Sailfish [39] use static analysis technologies to discover reentrancy vulnerability. In addition, there are dynamic testing and analysis tools such as ContractFuzzer [20], sFuzz [31], Smartian [7], RLF [44], and ReGuard [27], and approaches based on machine learning like ReVulDL [58]. As contract code and vulnerabilities become more complex, there are works that focus on cross-contract analysis, such as Clairvoyance [56] and SmartDagger [26]. However, many of these tools suffer from high false positive rates and may not identify real vulnerable contracts in practice [35].

To our knowledge, BlockWatchdog is the first detection tool to identify attacker contracts and their call chains. This feature enables the tool to detect real vulnerable victim contracts in practice. Additionally, BlockWatchdog addresses the limitation of current reentrancy detection capability [60], and can detect complex reentrancy attacks in the wild.

Security analysis on attack incidents on Ethereum. Prior research on attack analysis on Ethereum has focused on understanding attack incidents at the transaction level. For example, Torres et al. [47] conducted an empirical study of front-running attacks on Ethereum, while Zhou et al. [61] evaluated real-world attacks and defenses in the Ethereum ecosystem. They analyzed how attackers have destroyed applications in Ethereum and discussed how to defend against attacks from the victim contract's perspective. Su et al. [45] focused on DApp security and analyzed related transactions to understand how to detect attacks through transaction analysis. They developed a tool called DEFIER that can identify the stage of a potential attack. However, BlockWatchdog can detect attacker contracts without any attack transactions, making it possible to prevent financial loss.

8 CONCLUSION AND FUTURE WORKS

In this paper, we present BlockWatchdog, a tool for detecting reentrancy attacker contracts and identifying vulnerable victims with reentrancy vulnerabilities. To reduce false positives, we use the detection of attacker contracts as an entry point, and identify vulnerable victim contracts based on callback flow. To design Block-Watchdog, we conducted an empirical study to understand the attack logic used by hackers in attacker contracts.BlockWatchdog disassembles a contract's bytecode and monitors all potential call chains that initiate from its public functions and extend to accessible contracts and functions. Besides, BlockWatchdog formulates the xCFG and xCG to facilitate cross-contract data flow analysis between different procedures and to determine whether the callback flow can be exploited by malicious contracts to execute a successful reentrancy attack. Our experiment results demonstrate that Block-Watchdog effectively detects 113 attacker contracts among 421,889 real-world contracts and identifies 159 victim contracts with reentrancy vulnerabilities. These vulnerable contracts contain Ethers and tokens worth approximately 908.6 million USD. Only 18 of them are identified by other detection tools.

Furthermore, we reveal all potential call chains of the 421,889 real-world contracts, and whether they contain external calls in hook functions identified by BlockWatchdog. The detection results can be helpful for further security analysis. In the future, we plan to deploy BlockWatchdog for real-time detection purposes, in order to find more vulnerable contracts in practice and help prevent them from being attacked. In addition, we will extend BlockWatchdog to cover new types of attacks, remaining effective in the face of ever-evolving threats on Ethereum.

ACKNOWLEDGMENTS

REFERENCES

[1] 2022. Hacker drains $1.4 million worth of ETH from NFT lender Omni. https://www.theblock.co/post/156800/hacker-drains-1-4-million-worth-of-eth-from-nft-lender-omni Section: Hacks.

[2] 2022. Medium – Where good ideas find you. https://medium.com/

[3] alchemy 2023. alchemy. https://www.alchemy.com/.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[5] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*. IEEE, 85–103.

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[7] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239.

[8] consensys 2023. Consensys. https://consensys.net/.

[9] C.R.E.A.M. Finance 2023. C.R.E.A.M. Finance. https://docs.cream.finance/.

[10] EIP 2023. Ethereum Improvement Proposals. https://eips.ethereum.org/.

[11] eip-55 2023. EIP-55. https://eips.ethereum.org/EIPS/eip-55.

[12] ERC-3156: Flash Loans 2023. ERC-3156: Flash Loans. https://eips.ethereum.org/EIPS/eip-3156.

[13] erc1155 2023. ERC-1155 MULTI-TOKEN STANDARD. https://ethereum.org/en/developers/docs/standards/tokens/erc-1155/.

[14] erc777 2022. ERC-777 TOKEN STANDARD. https://ethereum.org/en/developers/docs/standards/tokens/erc-777/.

[15] etherscan 2023. Etherscan - The Ethereum Blockchain Explorer. https://etherscan.io/.

[16] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[17] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. 2022. Elipmoc: advanced decompilation of Ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–27.

[18] Harvard CodeBlue 2018. EIP-721: Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721.

[19] ivisor 2021. Visor Finance Suffers another DeFi Hack as Losses Mount Up to $8.2M. https://www.fxempire.com/news/article/defi-protocols-have-lost-680-million-so-far-in-2021-795829.

[20] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 259–269.

[21] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[22] Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. 2023. DeFiTainter: Detecting Price Manipulation Vulnerabilities in DeFi Protocols. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1144–1156.

[23] Johannes Krupp and Christian Rossow. 2018. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1317–1333.

[24] lendfme 2020. About Recent Uniswap and Lendf.Me Reentrancy Attacks. https://medium.com/imtoken/about-recent-uniswap-and-lendf-me-reentrancy-attacks-7cebe834ecb3.

[25] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. 2020. A survey on the security of blockchain systems. *Future generation computer systems* 107 (2020), 841–853.

[26] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 752–764.

[27] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 65–68.

[28] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[29] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam* 9 (2018), 54.

[30] Mythril 2023. Mythril. https://mythril-classic.readthedocs.io/en/master/module-list.html.

[31] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of*

the ACM/IEEE 42nd International Conference on Software Engineering. 778–788.

[32] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference.* 653–663.

[33] Robert Norvill, Beltran Fiz, Radu State, and Andrea Cullen. 2019. Standardising smart contracts: Automatically inferring ERC standards. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC).* IEEE, 192–195.

[34] peckshield 2023. Peckshield - Industry Leading Blockchain Security Company. https://peckshield.com/.

[35] Daniel Perez and Benjamin Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited.. In *USENIX Security Symposium.* 1325–1341.

[36] Phalcon 2023. Powerful Transaction Explorer Designed For DeFi Community. https://explorer.phalcon.xyz/.

[37] primitivefinance 2023. Whitehack by Primitive Finance: MOST FUNDS ARE SAFE. User action required. https://primitivefinance.medium.com/whitehack-by-primitive-finance-most-funds-are-safe-user-action-required-4dd31c387b8.

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

[39] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. 2012. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing.* 1–14.

[40] revest 2022. Revest Finance Vulnerabilities: More than Re-entrancy. https://blocksecteam.medium.com/revest-finance-vulnerabilities-more-than-re-entrancy-1609957b742f.

[41] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).

[42] Securify 2.0 2023. Securify 2.0. https://github.com/eth-sri/securify2.

[43] Donna Spencer. 2009. *Card sorting: Designing usable categories.* Rosenfeld Media.

[44] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2022. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering.* 1–12.

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

[46] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First monday* (1997).

[47] Christof Ferreira Torres, Ramiro Camino, and Radu State. 2021. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain. *arXiv preprint arXiv:2102.03347* (2021).

[48] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 67–82.

[49] Twitter 2023. Twitter. https://twitter.com/.

[50] UniswapV2 2023. UniswapV2. https://github.com/Uniswap/v2-core/blob/master/contracts/UniswapV2Pair.sol.

[51] uniswap_v3 2023. The Uniswap V3 Smart Contracts. https://docs.uniswap.org/contracts/v3/overvie.

[52] web3py 2023. web3py. https://web3py.readthedocs.io/en/stable/web3.eth.html.

[53] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. 2022. Sok: Decentralized finance (defi). In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies.* 30–46.

[54] whitehats 2023. Why White Hat Hackers Are Vital to the Crypto Ecosystem. https://www.coindesk.com/layer2/2022/02/23/why-white-hat-hackers-are-vital-to-the-crypto-ecosystem/.

[55] Kaidong Wu. 2019. An empirical study of blockchain-based decentralized applications. *arXiv preprint arXiv:1902.04969* (2019).

[56] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering.* 1029–1040.

[57] Shuo Yang, Jiachi Chen, and Zibin Zheng. 2023. Definition and Detection of Defects in NFT Smart Contracts. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023).* Association for Computing Machinery, New York, NY, USA, 373–384. https://doi.org/10.1145/3597926.3598063

[58] Zhuo Zhang, Yan Lei, Meng Yan, Yue Yu, Jiachi Chen, Shangwen Wang, and Xiaoguang Mao. 2022. Reentrancy vulnerability detection and localization: A deep learning based two-phase approach. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering.* 1–13.

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

[60] Zibin Zheng, Neng Zhang, Jianzhong Su, Zhijie Zhong, Mingxi Ye, and Jiachi Chen. 2023. Turn the Rudder: A Beacon of Reentrancy Detection for Smart Contracts on Ethereum. arXiv:2303.13770 [cs.SE]

[61] Shunfan Zhou, Zhemin Yang, Jie Xiang, Yinzhi Cao, Min Yang, and Yuan Zhang. 2020. An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem. In *Proceedings of the 29th USENIX Conference on Security Symposium.* 2793–2809.