Artifact Evaluated NDSS
Available
Functional
Reproduced

VETEOS: Statically Vetting EOSIO Contracts for the "Groundhog Day" Vulnerabilities

Levi Taiji Li
University of Utah
levili@cs.utah.edu

Ningyu He
Peking University
ningyu.he@pku.edu.cn

Haoyu Wang
Huazhong University of Science and Technology
haoyuwang@hust.edu.cn

Mu Zhang
University of Utah
muzhang@cs.utah.edu

***Abstract*—In this paper, we propose VETEOS, a static vetting tool for the *"Groundhog Day" vulnerabilities* in EOSIO contracts. In a *"Groundhog Day"* attack, culprits leverage the distinctive *rollback* issue in EOSIO contracts, which allows them to persistently execute identical contract code with varying inputs. By using the information exposed in prior executions, these attackers unlawfully amass insights about the target contract, thereby figuring out a reliable method to generate unauthorized profits. To tackle this problem, we formally define this unique vulnerability as a control and data dependency problem, and develop a custom static analysis tool, VETEOS, that can precisely discover such bugs directly from EOSIO WebAssembly (WASM) bytecode. VETEOS has detected 735 new vulnerabilities in the wild and outperforms the state-of-the-art EOSIO contract analyzer.**

I. INTRODUCTION

EOS [10] has become one of the major cryptocurrencies, with a market cap of 1.3 billion USD. Its official blockchain platform, EOSIO, provides an industry-leading transaction throughput – it processes around 4000 transactions per second (TPS), while Ethereum only has a transaction speed of 30 TPS. This is because EOSIO uses a "delegated proof-of-stake (DPoS)" [4] mining system rather than the traditional "proof-of-work (PoW)" [40] mechanism. Hence, EOSIO has the unique ability to enable high performance applications. EOSIO smart contracts, the building blocks of EOSIO applications, thus have attracted special attentions. Meanwhile, previous studies [18], [19], [7], [36], [30] have strived to comprehend and identify security vulnerabilities within EOSIO contracts.

In this paper, we propose VETEOS, a static vetting tool for the *Groundhog Day*[1] *vulnerabilities* (or GDV) in EOSIO contracts. In a *Groundhog Day* attack (or GHD attacks), adversaries can exploit the unique *rollback* problem [18] in EOSIO contracts to retry executing the same contract code repeatedly with different inputs. With leaked information observed during previous executions, attackers illegally accumulate knowledge about the victim contract, so as to learn how to make illicit profits in a deterministic manner. This kind of attacks may generally affect a broad spectrum of financial applications, such as sealed-bid auctions [39], double auctions [38], stock exchanges [6], casino number guessing game [18], etc., where internal contract states must be kept secret.

To our study, there exist four enabling factors in such an attack. *(F1) Revertable:* A sequence of activities in one EOSIO transaction can be reverted entirely by a malicious contract user. *(F2) Unpredictably profitable:* It is unpredictable what inputs can lead a user to make profits legally from a victim contract. *(F3) Information leakage:* A reverted transaction can change an internal state which however can be observed by external attackers. *(F4) Causal inference:* A leaked state can be leveraged by attackers to infer whether a specific input can make a profit.

Unfortunately, existing EOSIO contract analyzers [18], [7], [19] cannot completely or precisely detect such vulnerabilities, fundamentally due to their insufficient problem modeling. Prior work, such as EOSAFE [18], models this vulnerability largely as a "rollback" issue (*F1*). In contrast, it examines *F2* in an ad-hoc manner and only searches for special instances such as apps with random number generators. Furthermore, it does not discuss how internal contract states can be disclosed (*F3*) and does not adequately capture the causal relation (*F4*) between leaked information (*F3*) and profitability (*F2*).

To address this limitation, we propose to formally model a Groundhog Day vulnerability based upon these four critical factors. We further translate this abstract high-level model to concrete low-level detection specifications, and develop custom static analysis techniques to automatically identify such vulnerability patterns from EOSIO WebAssembly bytecode.

Our work makes three folds of contributions. First, we systematically studied real-world Groundhog Day problems in EOSIO contracts and gained an important insight: this class of vulnerabilities lies in intrinsic *data and control dependencies* among key contract constructs, such as user inputs, global states, database tables, API return values and inlined action calls. Second, we developed a custom static analysis tool that can perform context-sensitive flow-sensitive interprocedural dataflow analysis on EOSIO WebAssembly bytecode. To do so, we designed new algorithms to address *EOSIO contract-specific challenges* such as application-level entry points, indirect and implicit action calls, reordered dataflow due to delayed action execution and cross-action dataflow through database table accesses, and implemented new techniques to handle the distinct *memory model* and *calling convention*

[1]*Groundhog Day* is a 1993 film, where a weather reporter finds himself in a time loop on Groundhog Day; the day keeps repeating until he gets it right.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24972
www.ndss-symposium.org

```cpp
void apply(uint64_t receiver, uint64_t code, uint64_t
    action) {
  // action redirection
  if (action == name("func1").value) {
    eosio::execute_action(eosio::name(receiver), eosio::
      name(code), &mycontract::func3);
  }
  else if (action == name("func2").value) {
    eosio::execute_action(eosio::name(receiver), eosio::
      name(code), &mycontract::func4);
  }
  ...
}
```

Fig. 1: Example of `apply()` Function

used in EOSIO bytecode. Last but not least, we implemented a prototype, VetEOS, in 5,893 lines of Python code. We applied VetEOS to 60,577 real-world EOSIO contracts and discovered 735 novel vulnerabilities. Our evaluation showed that VetEOS significantly outperformed the state-of-the-art detector EOSafe, as VetEOS can reduce 79.8% of false positives that cannot be precisely recognized by the prior work. Our code, documents, and experimental data are publicly available at: https://github.com/HKJL10201/VetEOS.

II. Background

EOSIO Contracts. EOSIO contracts are computer programs running atop EOSIO blockchains. They are written in C++, compiled to WebAssembly (or WASM) bytecode, and running in EOSIO VMs, stack-based virtual machines. Unlike Ethereum contracts which are independent entities, an EOSIO smart contract must be associated with an EOSIO account and is automatically triggered when the account is invoked by another. When being triggered, an `apply()` function (exemplified in Figure 1) will be called. This function serves as a "dummy" starting point of an EOSIO contract and can further dispatch an external request to a specific function, called an *action*, based upon a given action name. Note that, the target action name does not have to match the requested one – the `apply()` function essentially performs a dictionary lookup and redirects the requested action to an existing function defined in the contract. For instance, in Figure 1, when the action `func1()` is requested, the `apply()` function will instead make a call to another action `func3()`.

Inline Action Sequence in One Transaction. Multiple actions can be performed in one single EOSIO transaction. This is achieved by making inline action calls. An inline action is in fact an implicit function call where contract and action names can be dynamically assigned at runtime. Figure 2 illustrates how to make an inline call to the `transfer()` action in the `eosio.token` contract so as to transfer tokens from a sender to a recipient. An inline action must be a tail call invoked at the end of another action, regardless where it is situated in a function. This is to ensure that all the statements in the previous action can be executed before moving to the next action. When a transaction is reverted, all the actions performed in this transaction will be reverted.

EOSIO Payment and Notification. Among all EOSIO contracts, `eosio.token` is a special system contract that allows users to create, issue, and manage tokens for EOSIO blockchains. For instance, one can check her own account balance by invoking the `eosio.token::get_balance()` function. Particularly, all funds (token) transfers in EOSIO

```cpp
void refund(eosio::name payee) {
  eosio::name code = name("eosio.token");
  eosio::name action = name("transfer");
  action(
    // permission level
    permission_level{get_self(), "active"_n},
    code,                      // target contract
    action,                    // target action
    std::make_tuple(           // transaction data
      get_self(),              // token sender
      payee,                   // token receiver
      asset(10000,             // token amount
        symbol("SYS", 4)),     // token symbol
      std::string("refund"))   // transaction memo
  ).send();
  ...
}
```

Fig. 2: Example of Inline Action Calls

systems must be realized using this contract, and therefore are essentially implemented in an asynchronous manner. To send tokens to a target account, an initiating account must make a call to the `eosio.token::transfer()` action, which will then update the balances of both accounts accordingly. Upon completion, this system contract will notify the sender and the recipient, and the notifications will be handled by the `apply()` functions in both contracts. This, however, leaves malicious users a window for launching the so-call "rollback" attacks [18]. Because the token transfer and the notification handling happen in the same transaction, a malicious notification receiver, can thus intentionally revert an already executed funds transfer – and potentially a series of previous actions in this transaction – by simply invalidating the notification via calling the `eosio_assert()` function in her `apply()` code.

EOSIO Table. The system contract `eosio.token` stores account balances in a persistent database storage. Such a database is accessible through the EOSIO `Table` interface. In fact, these database tables can be used by any EOSIO contracts to permanently maintain states even after the contract ends. This is a very useful feature for financial applications to synchronize states among multiple transactions. Although individual action calls (or transactions) are independent from each other and therefore by nature stateless, they all belong to the same stateful application logic and thus must share common data. Particularly, smart contracts often leverage tables to create and manage private accounts (as opposed to public accounts maintained by the system contract) and thus can allow users to directly work with the application contracts using their virtual accounts. Similar to any relational database tables, an EOSIO table is indexed by its primary key which must be defined when creating a table scheme. To access a table entry, a key is needed to reference its corresponding row.

WebAssembly Linear Memory Model. Another shareable data structure among action calls is the linear memory adopted by WebAssembly. Essentially, the linear memory is an array of bytes which can be accessed by any action calls. Actions use the linear memory to pass arguments and allocate local variables. In principle, this linear memory can also be used to store global variables for maintaining states across transactions. However, in practice, due to its volatile nature, it is unlikely that applications leverage the linear memory to persist long-term data such as account balances.

Authorization Check. Activities in EOSIO contracts, such as action calls, table accesses, can all be protected by authorization checks. An authorization check can be placed in any action or regular function to verify whether callers have sufficient permissions to make such a call. For instance, a critical action such as deleting accounts should be only invoked by its contract owner; then this action must be protected by this check: `require_auth(get_self())`. In the case where each user is allowed to access solely her own personal account, the action to read users' account balances must verify that the caller's identify matches that of the account owner.

Resource Model. The EOSIO resource model forms the backbone of resource allocation and management on the EOSIO blockchain platform. This model centers on three vital resources, CPU, NET and RAM, where CPU and NET are essential for executing transactions. Contract owners or transaction senders are required to stake EOS tokens to exchange these resources and determine the amount of CPU and NET resources at their disposal [22], [12]. Unlike the "gas" fees that are charged by Ethereum to execute individual instructions, staked tokens are not "spent" but are rather "locked" for a certain period. After this period, or once certain conditions are met, the tokens can be "unstaked" and returned to the user's control. Therefore, the staked tokens are reimbursable.

Secrecy in Smart Contracts. While blockchain data is inherently public, EOSIO contracts can incorporate methods to maintain secrets. *(1) In-memory Secrets.* Contracts can obtain unforeseen user inputs (e.g., bid prices) at runtime, or dynamically generate random numbers, leveraging blockchain properties, such as `tapos_block_num`, that are not easily predictable [17]. These secrets are used immediately within the contract code for condition checks and are not stored on the blockchain, preserving their confidentiality. *(2) Hashing Techniques.* Borrowing from the principles of password management, EOSIO contracts can store a hash of a secret rather than the secret itself on a blockchain (e.g., in EOSIO tables). It is virtually impossible to recover the original information from its hash. For instance, a sealed-bid auction can be conducted using the hashes of secret bids [29]. This process unfolds in two phases. During the "bidding" phase, each participant submits the hash of their bid along with a cryptocurrency deposit that matches or exceeds the value of the actual bid. Then, the later "revealing" phase allows participants to disclose their original bids. The integrity of these bids is confirmed by comparing them with the hashes submitted earlier.

III. Groundhog Day Vulnerability

A. Secret Revelation via Unlimited Free Trials

Many financial applications depend on time-sensitive secrets. These secrets cannot be disclosed before a transaction has completed. Otherwise, it will cause unfairness issues. For instance, in a blind auction (or first-price sealed-bid auction) [39], bidders only know whether they were the best but do not see others' bids. However, if the current highest bid is leaked to a bidder, she can thus maximize her profit, unfairly, by placing a bid that barely exceeds the previous one. Another example is the lottery game where players must guess a secret number to win. Similarly, such a game becomes compromised if the hidden number is revealed prematurely to any players.

To reveal such secrets, malicious contract users can simply perform black-box testing and eventually find the exact input that leads to expected profits. For instance, in a blind auction,

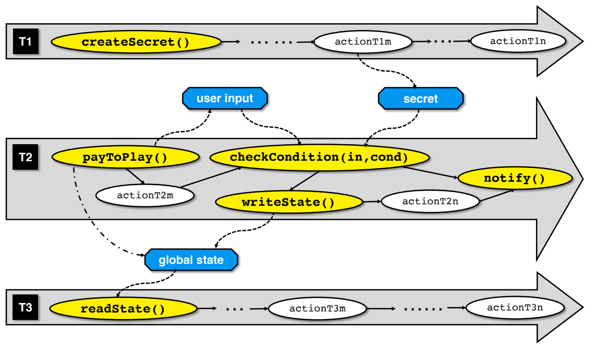


Fig. 3: Groundhog Day Vulnerability

a bidder can start with a high bid and gradually reduce her bid price to reach the "lowest" winning price. Note that, attackers do not need to understand the type of secrets they are probing. All they need is to pass the critical condition check against a secret and thus execute the "profitable" path.Normally, this brute-force attack is infeasible as every attempt has a non-negligible cost. However, because the "rollback" problem [18] in EOSIO contracts essentially allows unlimited free trials, such an exploit thus becomes possible. Of course, reverting a transaction alone does not sufficiently enable attackers to identify hidden secrets, since the outcome of the reverted transaction (i.e., whether a given input can make a profit) may not be revealed unless the transaction successfully completes. Hence, to uncover contract secrets during unlimited rollbacks, a Groundhog Day vulnerability must be in place.

B. Groundhog Day Attack Transactions

Figure 3 illustrates a typical Groundhog Day vulnerability. Here, solid lines represent control flows and curved dotted lines indicate dataflows. Ovals are actions or functions; octagons are global variables; gray arrows are transactions. At least three transactions are needed to enable this attack. First, a vulnerable contract must generate a secret via an earlier transaction *T1*. Such a secret can be derived in different ways. For instance, it can be directly created by the contract as a random number or a constant in a number guessing game; it can also be provided by a third party as an external user input – for example, the highest bid in a blind auction. Once a secret has been created, it can be directly used for comparison, or stored in a globally accessible region (e.g., linear memory or database tables) and later used by other actions to perform condition checks.

The second transaction *T2* is triggered and eventually reverted by an attacker to mount the attacks. It contains the core activities that enable threat actors to conduct black-box testing. Note that these activities can be implemented in either one or multiple EOSIO actions. Without loss of generality, in Figure 3, we illustrate them as several individual actions. This sequence of actions begins with an entry point function `payToPlay()` that allows a user to pay to participate in the contract activity such as gambling, auction, etc. This initial payment through a call to `eosio.token::transfer()` can cause changes to the user's account balance. While the payment has been accepted by the contract, the user's input (e.g., a lot, a bet or a bid) will also be received by the contract and stored in memory. Then, the user input will be compared with the previously generated secret by the `checkCondition(in,cond)`, and the result will be used to decide whether the "profitable" branch will be taken. If the "winning" condition is satisfied, the contract will need to update a global state via `writeState()`. This state can be the balance of the winner's official `eosio.token` account or her virtual account maintained locally by the contract via database tables. It may also be simply a global variable indicating the current winner. In the end, the contract notifies the participant (i.e., `notify()`) that the transaction has completed. By handling this notification, an attacker can intentionally rollback the entire transaction and decline all the actions in the *T2*.

Additionally, the attacker must execute the third transaction *T3* to observe the changes to the global state. This is critical because the result of T2 may be invisible to her since the transaction will not complete. Hence, this exposed information becomes the only chance for the attacker to understand whether her prior result has met her expectation and thus to adjust her strategy accordingly. Note that while reading a user's own account balance is in general allowed, accessing internal contract state such as the highest bidder may require certain permissions. In such cases, a GHD attack will only be possible if a necessary permission check, e.g., `require_auth()`, is accidentally missing or misused in the "readState" action. Also notice that, the timing at which the attacker checks the global state matters because the observed transaction *T2* may update the global state multiple times and only the specific state after the condition check can actually reflect its result. The attacker thus may need to check the global states several times and perform a differential analysis to detect any changes.

C. Formal Definition

Definition 1: A *Groundhog Day vulnerability* (GDV) in EOSIO contracts allows an attacker to indefinitely re-execute a transaction without any cost so that she can eventually identify the exact contract input that deterministically maximizes her profits. Hence, the existence of such a vulnerability depends on four key factors:

- (***F1***) **Revertable:** A sequence of activities locate in a single transaction that can be reverted entirely, so that a malicious user can rerun it *unlimitedly* for free.
- (***F2***) **Unpredictably profitable:** Whether one can make profits legally from a vulnerable contract is unpredictable. It relies on a *secret* condition the contract uses to evaluate participants' inputs.
- (***F3***) **Information leakage:** A state is changed in the middle of the revertable transaction. The state change is *visible* outside this transaction.
- (***F4***) **Causal inference:** The change to the visible state is caused by the invisible comparison between a user *input* and the secret, and can be used to infer the comparison result.

Based upon this definition, to determine whether an EOSIO contract is vulnerable to a Groundhog Day attack is equivalent to finding control and data dependencies that can fulfill the four requirements. For instance, in Figure 3, to satisfy *F1*, one must identify the inter-procedural control flow `payToPlay()` $\rightarrow$ `writeState()` $\rightarrow$ `notify()` in the transaction *T2*. In the meantime, *F2* indicates two inbound dataflows for the comparison in *T2*: `payToPlay()` $\rightsquigarrow$ *user input* $\rightsquigarrow$ `checkCondition()` and *T1* $\rightsquigarrow$ *secret* $\rightsquigarrow$ `checkCondition()`. *F3* requires a dataflow from *T2* to *T3*: `writeState()` $\rightsquigarrow$ *global state* $\rightsquigarrow$ `readState()`, while *F4* can be represented as the control dependency between `checkCondition()` and `writeState()`.

Algorithm 1 Detection of Groundhog Day Vulnerabilities

```
1: procedure ContainsGDV(c)
2:     $\mathbb{WR}$ ← FindWriteState(c)
3:     for ∀wr ∈ $\mathbb{WR}$ do
4:         gs ← GetGlobalState(wr)
5:         if IsTokenAccount(gs) or isReadable(gs, c) then
6:             ep ← FindEntryPoint(wr)
7:             if LeadToNotify(ep) then
8:                 in ← GetUserInput(ep)
9:                 $\mathbb{USE}$ ← DoDefUseChainAnalysis(in)
10:                for ∀use ∈ $\mathbb{USE}$ do
11:                    if IsCondition(use) and IsPred(use, wr) then
12:                        return true
13:                    end if
14:                end for
15:            end if
16:        end if
17:    end for
18:    return false
19: end procedure
```

Why Existing Work Fails. A Groundhog Day attack is built atop the rollback issues but is far more complex than a basic rollback attack. Fundamentally, this is because such attacks aim to make profits from vulnerable financial apps in a deterministic and general manner and therefore require a series of delicate program dependencies. Thus, while state-of-the-art analyzers [18], [7], [19] can effectively detect rollback issues, they cannot precisely or completely capture the GDVs.

IV. Detection Method

A. High Level Idea

At a high level, we detect the Groundhog Day vulnerabilities based upon the aforementioned control and data dependencies. Algorithm 1 illustrates our high-level idea. Particularly, given an EOSIO contract `c`, we first scan the entire contract to identify all the instructions $\mathbb{WR}$ that write global states. Then, for each instruction `wr` in this set, we obtain the global state `gs` it updates. We further check whether this state is also readable. This is equivalent to checking whether *(a)* this state is a user's `eosio.token` account balance that can be acquired by the user herself through a `get_balance()` call, or *(b)* it is contract-wide global variable (often a database table) that can be readable through an external interface within the contract.

If either condition is satisfied, this global state can be leaked at runtime. We thus investigate whether the leaked state can be leveraged by adversaries to find a good input. To this end, we search for the entry point `ep` of the `wr` instruction. Such an entry point must be an action that allows users to start a transaction with user-specified inputs and that eventually notifies users of the result of a game. Hence, from the entry point, we can obtain the user input `in` and perform def-use chain analysis to discover all the uses $\mathbb{USE}$ of this input. We then examine each `use` statement in the $\mathbb{USE}$ set. If it is a conditional statement and meanwhile a predecessor of the state update `wr`, that means the changes to the global state are actually caused by user inputs, and therefore the leakage of this state information is indeed helpful to attackers. As a

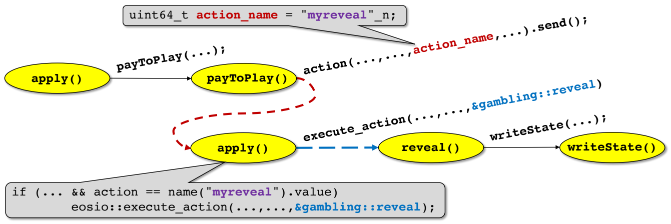
```
 1 class [[eosio::contract]] gambling:public eosio::contract{
 2 public:
 3   using eosio::contract::contract;
 4   [[eosio::action]] void reveal(eosio::name username, std
       ::string user_input) {
 5     action(permission_level{"gambling"_n, "active"_n},
 6       "gambling"_n, "getbalance"_n,
 7       std::make_tuple(username)).send();
 8     ...
 9     uint64_t secret = getSecret();
10     if (checkCondition(user_input, secret)) {
11       writeState(username, state);
12       createSecret(); ...
13     }
14     notify(username, message);
15   }
16   [[eosio::action]] void getbalance(eosio::name username){
17     require_auth(get_self());
18     uint64_t balance = readState(username);
19     ...
20     notify(username, message);
21   }
22   [[eosio::action]] void myreveal(eosio::name username,
       std::string user_input) { ... }
23 private:
24   uint64_t getSecret();
25   void createSecret();
26   bool checkCondition(std::string user_input, uint64_t
       secret);
27   void writeState(eosio::name username, uint64_t state);
28   uint64_t readState(eosio::name username);
29   void notify(eosio::name username, std::string message);
30   ...
31   using balance_index = eosio::multi_index<"balances"_n,
       balance>;
32 };
33
34 void payToPlay(const transfer_data &transfer) {
35   ...
36   eosio::name contract_name = name("gambling");
37   eosio::name action_name = name("myreveal");
38   action(permission_level{"gambling"_n, "active"_n},
39     contract_name, action_name,
40     std::make_tuple(transfer.from, transfer.memo)).send();
41 }
42
43 extern "C" void apply(uint64_t receiver, uint64_t code,
      uint64_t action) {
44   if (code==receiver && action==name("myreveal").value)
45     eosio::execute_action(eosio::name(receiver), eosio::
        name(code), &gambling::reveal);
46   else if (action == name("getbalance").value)
47     eosio::execute_action(eosio::name(receiver), eosio::
        name(code), &gambling::getbalance);
48   else if (code == name("eosio.token").value && action ==
        name("transfer").value)
49     payToPlay(unpack_action_data<transfer_data>());
50 }
```

Fig. 4: An Example of Gambling Contract

result, we can return true at this point. Eventually, if we do not identify any “true” patterns, we return false.

B. Technical Challenges

While the high-level detection algorithm is straightforward, several technical challenges must be addressed. To explain these challenges and our solutions, we present a concrete example as depicted in Figure 4. This example illustrates an EOSIO contract implementing a gambling game. This code allows any player to place a bet in a number guessing game via sending EOS tokens to the contract's account using `eosio.token::transfer()` (LN48). Upon a successful funds transfer, the `eosio.token` contract will send a notification to this contract, which will be handled by the `apply()` function (LN43). This then triggers the `payToPlay()` function (LN34) which will further call the `reveal()` action (LN4)



```
uint64_t action_name = "myreveal"_n;
```

```
if (... && action == name("myreveal").value)
        eosio::execute_action(...,...,&gambling::reveal);
```

apply()
payToPlay(...);
payToPlay()
action(...,...,action_name,...).send();
apply()
execute_action(...,...,&gambling::reveal)
reveal()
writeState(...);
writeState()

Fig. 5: Attack Flow in the Gambling App

to check the player's bet against the predefined “secret”. If the two numbers match, the contract will update the player's “status” – e.g., sending the winner's prize to her account. Regardless of the player's outcome, the contract will send a notification to inform her of the result.

Note that, while we aim to automatically analyze EOSIO WASM bytecode, for the readability purpose, we present the contract in C++ source code. Particularly, this contract contains a “dummy” entry point function `apply()` (LN43), an internal function `payToPlay()` (LN34) as an event handler for inbound token transfers, three public actions: `reveal()`, `getbalance()` and `myreveal()` (LN4-22), and several private member functions including `getSecret()`, `createSecret()`, `checkCondition()`, `writeState()`, `readState()` and `notify()` (LN24-29). Actions can be invoked directly by external parties while private functions can only be triggered internally. The contract also maintains a contract-wide database `balances` (LN31) accessible by the `writeState()` and `readState()` functions (LN27-28), which are further illustrated in Figure 6.

By applying Algorithm 1, we hope to first identify the call to the `writeState()` function (LN11) which changes the global `balances` state. Starting from this call, we trace back to discover its entry point: the if-clause for `payToPlay()` in the `apply()` where every player's input – including the sender (e.g., `from`) and recipient (e.g., `to`) of a funds transfer, the transferred token `quantity` and the additional `memo` (e.g., the guessed number) – is encoded in a `transfer_data` (LN49). Then, from this entry point, we perform a forward data and control-flow analysis to find that the user input (`transfer.memo`) is being checked against a potential secret (LN10) to determine whether the player wins. If so, the player's account status will be updated by the `writeState()`. In the meantime, the state change can be leaked through a call to `readState()`, which in this case is automatically invoked (LN5) each time a new bet is placed.

Nevertheless, to achieve the expected analysis result, several unique challenges must be addressed.

C1: Correctly identifying application entry points so as to capture required vulnerability factors.

Entry point identification plays a crucial role in revealing potential attack flows in EOSIO apps. For instance, in the motivating example, we may detect the existence of the Groundhog Day vulnerability, only if we can correctly recognize the entry point of the function for updating account states (i.e., `writeState()`) to be the function `payToPlay()`, as illustrated in Figure 5. This is because `payToPlay()` is a necessary step to satisfy two required vulnerability factors: *(a)* it accepts players' bets and thus makes the contract *unpredictably* *profitable* (*F2*); *(b)* it initializes players' account states based upon their transferred funds, and therefore allows attackers to later observe possible state updates (*F3 Information leakage*).

C2: Soundly tracking control flows through implicit and indirect calls to complete callgraphs.

To find the entry point of `writeState()`, we need to perform callgraph analysis. In this case, eventually, we expect to identify this call chain: `apply()` → `payToPlay()` → `apply()` → `reveal()` → `writeState()`. As illustrated in Figure 5, this call chain contains direct calls (solid arrow), indirect calls (blue dotted arrow) and implicit calls (red dotted arrows). Specifically, the indirect call from `payToPlay()` to `apply()` is triggered via passing a function reference `&gambling::reveal` to the `eosio::execute_action()` API. Meanwhile, the `reveal()` action is implicitly invoked by the `apply()` function in a "reflective" manner – an `action_name` string is fed into an inline action call `action().send()` as an argument.

However, traditional callgraph analysis only discovers explicit caller-callee relations but does not identify these indirect or implicit calls. If either of the two links is broken, one will fail to uncover the causal relation among `payToPlay()`, `reveal()` (which calls `checkCondition()`) and `writeState()` that fundamentally forms a Groundhog Day bug. Furthermore, to accurately address implicit inline action calls, it is also crucial to correctly handle the unique *dynamic dispatch* mechanism used in the custom `apply()` function. As shown in Figure 5, while the `reveal()` action is invoked by an inline call from `payToPlay()`, the `action_name` specified in the call is actually not "reveal" but "myreveal". It is the handler code in the `apply()` that dispatches the "myreveal" request to the `&gmabling::reveal` reference. In fact, there is indeed an action `myreveal()` in this contract. Without interpreting the dispatcher code, a callgraph analysis will miss this redirected call to the `reveal()` action.

C3: Accurately establishing data dependencies due to delayed execution of inline action calls.

Regardless of where an inline action is being invoked within a function, it will be executed last, following the execution of all other statements in the function. Unfortunately, conventional dataflow analysis does not recognize such a reordering in the control flow, and therefore fails to correctly discern the data dependencies between the inline call and the remaining components of the function. For instance, in the `reveal()` action (Figure 4), the delayed execution of the inline call to `getbalance()` in effect allows it to access the newer account state which may have been updated by `writeState()`. However, because this inline call *seems* to happen before `writeState()` from a traditional control-flow perspective, existing dataflow analysis techniques – without specifically handling the delayed inline call – will mistakenly consider that users cannot obtain their updated account states from this call and therefore cannot infer whether their inputs have passed the condition check. Note that, the `getbalance()` action cannot be called directly by any arbitrary users because it is protected by a strict permission check that requires the caller to be only the contract itself. As a result, this `getbalance()` inline call within the `reveal()` action, invoked by the contract, is the only opportunity for attackers

```
void writeState(eosio::name username, uint64_t state) {
  ...
  balance_index balances(get_self(), get_self().value);
  auto iterator = balances.find(username.value);
  if (iterator == balances.end()) {
    balances.emplace(get_self(), [&](auto &row) {
      row.key = username;
      row.amount = amount; });
  }
  else {
    balances.modify(iterator, get_self(), [&](auto &row) {
      row.key = username;
      row.amount = amount; });
  }
}
uint64_t readState(eosio::name username) {
  balance_index balances(get_self(), get_self().value);
  uint64_t amount = 0;
  auto iterator = balances.find(username.value);
  if (iterator != balances.end())
    amount += iterator->amount;
  return amount;
}
struct [[eosio::table]] balance {
  name key;
  uint64_t amount;
  uint64_t primary_key() const { return key.value; }
};
using balance_index = eosio::multi_index<"balances"_n,
     balance>;
```

Fig. 6: `writeState()` and `readState()`

to read their account states. Subsequently, the misconception that this inline call is unable to retrieve account updates can result in a detector overlooking this vulnerability.

C4: Tracking dataflow across actions via global database table accesses.

EOSIO contracts use persistent storage, database *tables*, to maintain and share contract-wide states. Hence, we must capture dataflow through these unique programming constructs. Figure 6 illustrates an example where `writeState()` and `readState()` have data dependencies due to a shared EOSIO table. In particular, this code declares a *multi-index* table, as `balance_index`. This table is named "`balances`" and configured to use the `balance` data structure. The `balance` struct contains two fields: an EOSIO `name` object `key` and an unsigned integer-typed `amount`. A `primary_key()` function is also defined for this struct to identify the key of the database table. Consequently, `balance_index` can be used to create references to the table that maps users to their account balances. Then, when accessing the table, one needs to use the `find(key)` API to locate the corresponding entry and use the `iterator` API to read (e.g., `iterator→amount`) or write (e.g., `modify(iterator,...)`) the entry. Hence, to precisely detect the dataflow through an EOSIO table, we must verify that multiple table access operations – such as `table1.find(key1)` and `table2.find(key2)` – read/write the same database *table* and the identical table *entry*.

C. Analysis Method

To address the aforementioned challenges, we develop our custom entry point discovery and dataflow analysis techniques.

a) Entry Point Discovery: Entry point identification has been a well-studied problem for event-driven programs such as Android apps [23] or industrial controller routines [41]. For instance, the state-of-the-art work CHEX [23] proposes to model an entry point of an Android program as a method that is not being called internally. Conceptually, we can follow the same idea to detect entry points of EOSIO apps. Nevertheless, in practice, we must precisely and completely identify the unique mechanisms, in the new context EOSIO contracts, that are used to make function calls. Particularly, in conventional event-driven programs, externally-facing functions such as GUI event handlers are typically not invoked from within application code and thus can be easily identified as entry points. In contrast, external interfaces in EOSIO smart contracts – *actions* – can still be called internally in an either direct, indirect or implicit manner. Hence, it is inadequate to simply consider external-facing actions to be entry points. In contrast, an internal function in EOSIO contracts may also serve as an entry point, as long as it is only invoked by `apply()` due to an `eosio.token::transfer()`. In fact, in the motivating example, if we mistakenly treat the `reveal()` action, instead of `payToPlay()`, as the entry point of `writeState()`, our vulnerability analysis will miss the critical factors that can only be found in `payToPlay()`.

Consequently, to capture the entirety of potential attack flows in an EOSIO application and thus be able to discover sufficient vulnerability patterns, we must precisely define an entry point for an EOSIO contract. Such an entry point must indicate the starting point of the *application business logic*, such as the `payToPlay()` of a gambling game, rather than an intermediate step (e.g., the `reveal()` function):

Definition 2: An *entry point* of an EOSIO application is either an EOSIO action that is not invoked directly, indirectly or implicitly by any other functions or actions in the same app, or an internal function that is solely triggered by the `apply()` function to handle the token transfers from the system contract.

Based upon this definition, to identify entry points, we must discover and inspect all types of action calls in addition to traditional callgraph analysis. To this end, we have summarized the different mechanisms that can be used to make action calls, as depicted in Table I. In general, an EOSIO action can be *(1)* called directly – the same way as regular function are called, *(2)* invoked indirectly using a function reference, or *(3)* triggered implicitly in a "reflective" manner. Any action call may be redirected to a concrete target via a *(4)* dispatcher handler. Thus, we must handle individual methods of action calls differently. While direct call targets can be simply resolved using conventional callgraph analysis, we perform custom pointer analysis to identify the targets of indirect calls, and use string analysis to interpret implicit "reflective" calls as well as connecting an implicit caller to an actual callee.

Our custom pointer analysis is based upon the *call_indirect* mechanism used in EOSIO bytecode. This special instruction makes an indirect call according to its integer argument, representing an index of a function. We perform backward dataflow analysis at each callsite to identify a constant source of its index value, and check the function table stored in the WASM file to determine the call target. Our string analysis follows Christensen et al.'s classic algorithm [8]. The general idea is that a series of string operations can be translated into an automaton and possible string values at a certain point of interest – e.g., the action name being called – must be accepted by this automation. More concretely, starting from a string initialization (e.g., constant) of interest, we first build a *flow graph* [8] that captures data dependencies among string operations in the bytecode. Next, we transform the graph to a context-free grammar, and utilize the Mohri-Nederhof [25] algorithm to approximate this grammar with a regular grammar, and finally extract automata from the latter. Our analysis handles major string operations, defined either in `eosio::string` or the standard C++ library `std::string`, including `append()`, `insert()`, `substr()` and `replace()`.

Algorithm 2 Entry Point Discovery

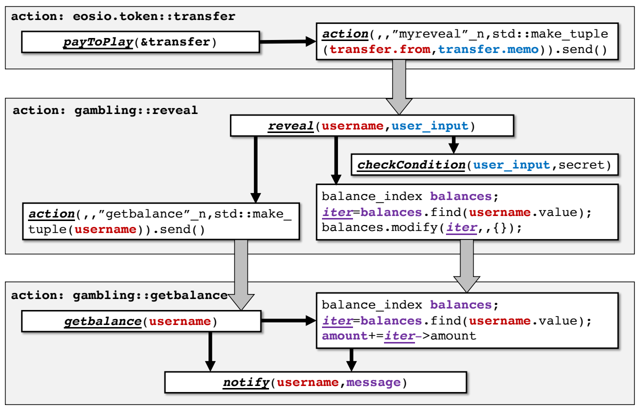
```
 1: procedure FINDENTRYPOINTS(poi)
 2:     EP ← ∅
 3:     Q ← ∅
 4:     Visited ← ∅
 5:     Q.ENQUEUE(HOSTFUNC(poi))
 6:     while Q.ISNOTEMPTY() do
 7:         f ← Q.DEQUEUE()
 8:         Visited ← Visited ∪ f
 9:         Clr ← (DIRCLR(f) ∪ INDIRCLR(f) ∪ IMPCLR(f)) − Visited
10:         if Clr = ∅ then
11:             EP ← EP ∪ f
12:         else if Clr = {apply(, eosio, transfer)} ∧ !ISACTION(f) then
13:             EP ← EP ∪ f
14:         else
15:             Q.ENQUEUE(Clr)
16:         end if
17:     end while
18:     return EP
19: end procedure
```

TABLE I: Action Call Methods

Category	Example	Required Analysis
Direct call	`reveal();`	Callgraph analysis
Indirect call	`execute_action(...,...,&gambling::reveal);`	Pointer analysis
Implicit call	`action(...,...,‘‘myreveal’’_n,...).send();`	String analysis
Dispatching	`if(action==name(‘‘myreveal’’).value) execute_action();`	String analysis

With our pointer and string analyses, we can discover complete caller-callee relations in an EOSIO contract and address *C2*. Then, we develop the algorithm to explore the entry points for a given point of interest in a contract, as shown in Algorithm 2. This algorithm `FindEntryPoints()` takes a point of interest `poi` as an input and discovers all the contract entry points `EP` that lead to this point. It first initializes three empty sets: the entry point set `EP`, a work queue `Q` and a set of visited functions `Visited`. Next, it inserts the host function of `poi` into `Q` and starts to process every function in the queue until the queue becomes empty. In each iteration, we fetch one function `f` from the queue and add it to the `Visited` set. Then, we compute the unvisited direct callers `DirClr()`, indirect callers `IndirClr()` and implicit callers `ImpClr()` of the function `f`. If the result `Clr` is an empty set, the current `f` is then identified as an entry point and added to `EP`. Or, if `Clr` contains only the `apply()` function triggered by `eosio.token::transfer()`, and `f` is not an action, we also consider `f` to be an entry point. Otherwise, we add `Clr` into the work queue for further processing. Eventually, when no new function need to be examined, we reach a fixed point and output `EP`. Thus, *C1* is finally addressed.

b) Cross-Action Dataflow Analysis: The unique challenges for analyzing dataflow in EOSIO contracts originate from the *special interactions among actions*. While dataflow within each action can be adequately addressed by classic def-use chain analysis, capturing the dataflow across multiple actions requires accurate modeling of action ordering. This determines whether and how information can flow from one action to another through a shared database table.



```
action: eosio.token::transfer
payToPlay(&transfer)
action(,,"myreveal"_n,std::make_tuple
(transfer.from,transfer.memo)).send()

action: gambling::reveal
reveal(username,user_input)
checkCondition(user_input,secret)
action(,,"getbalance"_n,std::make_
tuple(username)).send()
balance_index balances;
iter=balances.find(username.value);
balances.modify(iter,,{});

action: gambling::getbalance
getbalance(username)
balance_index balances;
iter=balances.find(username.value);
amount+=iter->amount
notify(username,message)
```

Fig. 7: Cross-Action Dataflow

Existing work that aims to analyze cross-"component" dataflow such as CHEX [23] takes a simplistic approach to model the interactions among external-facing components. Because prior work assumes that user-facing components are independent from each other, it proposes a random permutation-based dataflow analysis – it first computes a dataflow summary for each independent component and then checks different combinations of the components to find potential cross-component dataflows. This simple model, however, does not sufficiently address the interactions of actions in EOSIO contracts. This is because not all combinations of actions are viable due to the control-flow constraints caused by inline action calls. These constraints may actually affect the existence of data dependencies. Suppose a `readState()` function can only be called *before* a critical state change made by `writeState()`, and therefore cannot receive the updated state data. In such a case, overlooking this control dependency, one may mistakenly conclude that there exists a dataflow from `writeState()` to `readState()`.

Hence, we propose a model-constrained permutation-based cross-action dataflow analysis. Specifically, we first perform intra-procedural dataflow analysis within each action and summarize the result as a sources-to-sinks mapping. Then, we use our aforementioned callgraph analysis to construct an *action-flow model* that represents the partial order of action calls. Finally, guided by this model, we combine multiple dataflow summaries via connecting one's sink to another's source, so as to uncover information flow across actions. Figure 7 illustrates our analysis result for the motivating example.

Dataflow Summary. For every action, we use classic def-use chain analysis to identify its internal data dependencies between sources and sinks, and generate a *dataflow summary*. In particular, we have identified two types of sources: action inputs and database tables, and three types of sinks: inline action calls, database tables and signature components that are part of the Groundhog Day vulnerability. In Figure 7, for instance, we discover one source and three sinks in the `gambling::reveal` action. The source is the action interface `reveal(username,user_input)` which can receive two external inputs. This source data can flow into three different sinks: *(1)* an inline call to the action `getbalance()` which uses the tainted `username`, *(2)* a database write that `modif[ies]` a specific table entry indicated by the `username`, and *(3)* the `checkCondition()` component,

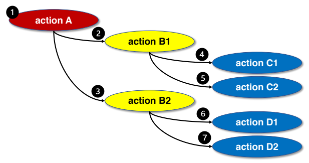


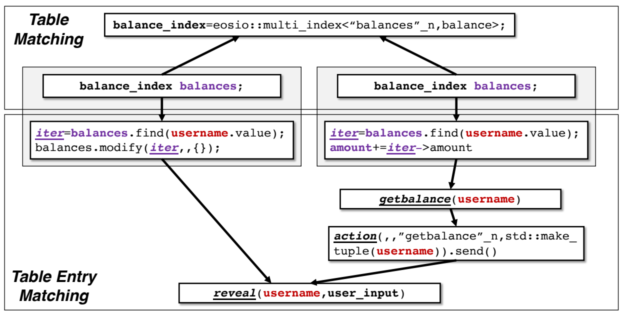
Fig. 8: Nested Inline Action Calls

an essential part of the vulnerability, which compares `username` with a predetermined secret. Then, a dataflow summary indicates any possible mappings from a source to a sink such as `reveal(username,user_input)` $\rightsquigarrow$ `action(,,``getbalance''_n,std::make_tuple(username)).send()`. Note that a table read (e.g., `iter→amount` in the `gambling::getbalance` action) can be both a source and a sink – it is the sink of the table key, `username`, within the action, and the source of the table content obtained from other actions.

Action-flow Model. In principle, any EOSIO actions can be directly called by external users, and therefore the order of action calls is indeterministic. However, in practice, due to certain restrictive permission requirements – e.g., the `require_auth(get_self())` in the `getbalance()` action – a seemingly external-facing action may in practice be only triggered internally by its host contract. Consequently, the control dependency between such an action and its caller then becomes deterministic and is constrained by how it is being called programmatically. In the motivating example, for instance, there is no way for third-party users including attackers to call `getbalance()` before invoking `reveal()`. To rule out impossible combinations of action calls in further permutations, we must model deterministic control flows.

Definition 3: An action-flow model of an EOSIO contract describes the *must-follow relations* among action calls. If action A *must follow* action B, all activities within A *must inherently follow* any activities within B, due to the delayed execution of A.

The *must-follow* relations result from inline action calls that can only be made by their host contract code (due to stringent permission checks). To capture such relations, we first search for restrictive permission checks (e.g., `require_auth(get_self())`) in each action to identify anyone that cannot be invoked by a third party. Then, for the identified ones, we use inter-procedural control-flow analysis to discover by whom and how they are being called. Therefore, a *must-follow* relation can be easily determined in two simple scenarios: *(a)* if action `X` simply calls `Y`, then `Y` *must follow* `X`; *(b)* if two actions `Y1` and `Y2` are being called sequentially by the same caller `X`, then `Y2` *must follow* `Y1`. However, the execution order may become less obvious when inline action calls are made in a nested fashion. Figure 8 gives an example. Here, `action A` makes a call to `action B1` and `action B2` consecutively; `B1` and `B2` then invoke `C1`, `C2` and `D1`, `D2`, respectively. While the order regarding direct caller/callee (e.g., `B1` and `C1`) and direct siblings (e.g., `B1` and `B2`) is clear, the order regarding `B2` and `C1` remains ambiguous. In fact, to the best of our knowledge, in what order nested inline calls are executed is not well documented. To address this question, we conduct an empirical study and discover that, when handling nested actions, EOSIO VM takes a breadth-first search-based



```
balance_index=eosio::multi_index<“balances”_n,balance>;
balance_index balances;
balance_index balances;
iter=balances.find(username.value);
balances.modify(iter,,{});
iter=balances.find(username.value);
amount+=iter->amount
getbalance(username)
action(,,”getbalance”_n,std::make_tuple(username)).send()
reveal(username,user_input)
```

Fig. 9: Two-Level Matching for Table Accesses

approach and processes the calls at the same level first before advancing to the next level. As a result, the total order of these seven action calls will be `A` ⇝ `B1` ⇝ `B2` ⇝ `C1` ⇝ `C2` ⇝ `D1` ⇝ `D2`. This action-flow model thus addresses *C3*.

Model-Constrained Permutation. We then use our action-flow model to guide the permutation of individual dataflow summaries. To do so, we first select two random actions `A` and `B`, and connect `A` to `B`. Next, we check whether such a connection violates any *must-follow* constraints, and if so, we discard it. Otherwise, we will search for any possible linkage between `A`'s sinks to `B`'s sources. A successfully discovered linkage indicates a cross-action dataflow. For instance, to generate the result in Figure 7, we first build the connection from the action `reveal` to `getbalance`. This connection satisfies the *must-follow* constraints which require the former to always occur before the latter. Consequently, we can test whether the two sinks of `reveal` can be linked to the two sources of `getbalance`.

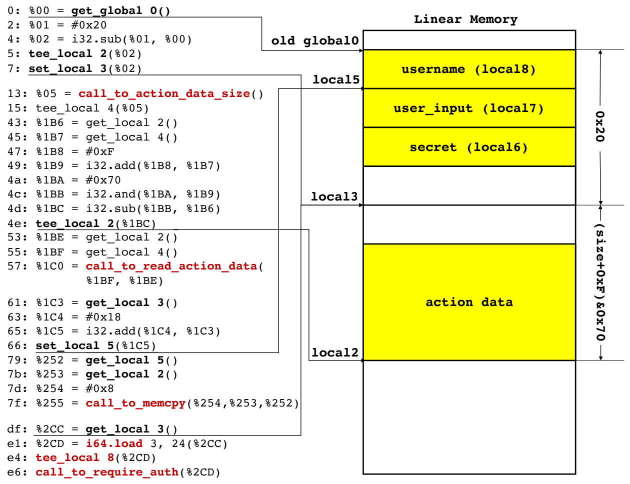
A cross-action dataflow linkage can be established due to two reasons. First, an inline action caller can be linked to a corresponding callee – for example, `action(,,‘‘getbalance’’,std::make_tuple(username)).send()` ⇝ `getbalance(username)`. Then, the dataflow is natually identified through the argument passing. Second, a “write” to a global database table is connected to a later “read” from the same table. For isntance, in Figure 7, we link `balances.modify(iter...)` to `amount+=iter→amount`. To determine whether two table accesses reference the same table entry, we conduct a two-level matching: table matching and table entry matching, as illustrated in Figure 9. *(1)* A table matching applies backward dataflow analysis to the table instances that are instantiated at different locations. If multiple instances originate from the same table object (e.g., `balance_index` in the case), a match is found and we can then proceed to the table entry matching. *(2)* The table entry matching also uses backward dataflow analysis. It aims to discover the origin of the accessed table keys. In this example, because the keys accessed at two different places share the same source (i.e., the `username` parameter from the `reveal()` call), we can conclude that there exists a dataflow between the two operations. Hence, *C4* is also addressed.

V. IMPLEMENTATION

Our dataflow analysis is performed on EOSIO WASM bytecode. We build our analysis on top of Octopus [1]. We fisrt leverage Octopus to convert WASM stack-based bytecode to

```
[[eosio::action]] void checkCond(eosio::name username,
    uint64_t user_input, uint64_t secret) {
  require_auth(username);
  if (user_input == secret)
    eosio::print(username, "wins");
  else
    eosio::print(username, "loses");
}
```

Fig. 10: Example Action Using Arguments



```
0: %00 = get_global 0()
2: %01 = #0x20
4: %02 = i32.sub(%01, %00)
5: tee_local 2(%02)
7: set_local 3(%02)

13: %05 = call_to_action_data_size()
15: tee_local 4(%05)
43: %1B6 = get_local 2()
45: %1B7 = get_local 4()
47: %1B8 = #0xF
49: %1B9 = i32.add(%1B8, %1B7)
4a: %1BA = #0x70
4c: %1BB = i32.and(%1BA, %1B9)
4d: %1BC = i32.sub(%1BB, %1B6)
4e: tee_local 2(%1BC)
53: %1BE = get_local 2()
55: %1BF = get_local 4()
57: %1C0 = call_to_read_action_data(
          %1BF, %1BE)

61: %1C3 = get_local 3()
63: %1C4 = #0x18
65: %1C5 = i32.add(%1C4, %1C3)
66: set_local 5(%1C5)
79: %252 = get_local 5()
7b: %253 = get_local 2()
7d: %254 = #0x8
7f: %255 = call_to_memcpy(%254,%253,%252)

df: %2CC = get_local 3()
e1: %2CD = i64.load 3, 24(%2CC)
e4: tee_local 8(%2CD)
e6: call_to_require_auth(%2CD)
```

Fig. 11: Memory Model and Calling Convention

static single assignment (SSA)-formed, register-based IR, and then conduct context-sensitive flow-sensitive interprocedural dataflow analysis on this IR. Nevertheless, while Octopus handles basic WebAssembly features, it does not specifically address the special memory model of EOSIO WASM bytecode caused by the distinct compilation process adopted by the EOSIO toolchains. Thus, we must implement custom dataflow analysis techniques to address the unique memory addressing mode and calling convention in EOSIO WASM code.

Basic WASM bytecode uses a simple memory model – it leverages special variables called *locals*, such as `local0`, `local1`, to store temporary data and pass function arguments. However, due to the usage of a linear memory and the special action argument passing, EOSIO WASM code may not directly use *locals* to transfer parameter data. On top of the basic simple model, it additionally adopts an indirect memory addressing mode. Figure 11 demonstrates this unique memory modeling and how action parameters are passed using this model.

Figure 11 illustrates the memory layout, when an EOSIO action `checkCond()` is called, and partial SSA-formed Octopus IR code that allocates and manages this memory space. The source code of this action is presented in Figure 10. This action implements a simplified versoin of the `checkCondition()` function in the motivating example. It takes three arguments: the `username` who makes the call, a `user_input` that indicates a guessed number, and the `secret` number generated by the contract. The action uses the `username` for the authorization check, and then compares `user_input` with `secret` to determine if the user's guess is correct, and finally presents the result to the user.

Nevertheless, our study on the IR instructions (Figure 11) indicates that these three action arguments are not stored in any WASM *locals*. In fact, the *locals* here hold addresses rather than values and are thus used to reference memory regions that contain these parameters. Similar to a stack pointer in x86, a special `global0` variable is used in this model to always point to the bottom of the memory being used. As a result, at the start of the `checkCond()` function, the `global0` pointer is moved to a lower position to allocate necessary spaces for arguments and local variables. More concretely, this IR code first allocates a block of `0x20` bytes and saves its address in `local3`. Next, it creates additional space based upon the argument size of this action. To do so, it invokes the `call_to_action_data_size()` to obtain the raw data size of action input, and computes the actual storage space (`(action_data_size + 0xF) & 0x70`) by considering the alignment requirement, and finally reserves a memory block starting at the address in `local2`. Then, the `call_to_read_action_data()` function is called to pass the action arguments to the reserved memory space pointed to by `local2`. To store the arguments into local variables for later usage, the code further calls `call_to_memcpy()` to transfer individual parameter values to corresponding positions within the memory region at `local3` (e.g., `local5` for `username`). To easily reference these local variables, their values are eventually loaded into other *locals* such as `local8`, `local7` and `local6`. For instance, the `local8` (i.e., `username`) can be directly used as a parameter in the `require_auth()` call.

Custom SSA transformation to handle the indirect memory addressing mode. Existing SSA transformation in Octopus is only applied to simple IR variables (stored in WASM stack) but not WASM *locals* because these *locals* are, technically, not "redefined" in the code. Consequently, even though the content of a *local* has been altered, its name does not change. For instance, in Figure 11, there exist two memory writes that change the `local2`'s content: `tee_local2(%02)` (LN5) and `tee_local2(%1BC)` (LN4e). However, any accesses to this *local* (e.g., LN43, LN53, LN7b) refer to it as the same name "`local2`". Hence, when the value of `local2` is used as an address from which memory content is being loaded (e.g., `memcpy()`), it is not explicit what data will be obtained. To eliminate this confusion, we additionally apply SSA transformation to the *locals* and rename them whenever their contents are modified by write operations such as `set_local` or `tee_local`. Thus, in this case, we will have two different versions of `local2`. Then, to further track dataflow through memory data loaded from addresses specified by *locals*, we develop a custom points-to analysis atop renamed *locals*. Specifically, for the sake of efficiency, we use a strict policy to identify aliases. For two (*local* + constant offset) patterns, we consider they are aliases only if the two *locals* share the same data origin and the two offsets are identical. In theory, our approach may lead to incompleteness. We will assess the accuracy of our dataflow analysis in the evaluation.

Using signature functions to recognize action arguments. To our study, the action arguments are transferred using two signature functions `call_to_action_data_size()` and `call_to_read_action_data()`. The return value of the former will be used as one parameter of the latter. The other parameter of the latter is the address of the transferred arguments. Therefore, we leverage the existence of the two signature functions and their data dependencies to identify the positions of action parameters.

VI. Evaluation

A. Experimental Setup

Three Datasets. *(a)* To assess the accuracy of our bytecode analysis, we must leverage source code-level information as the ground truth. To this end, we have retrieved 98 real-world EOSIO smart contracts, from open-source projects on GitHub, whose source code can be identified. We further compile these projects to generate corresponding bytecode programs.

(b) To discover new vulnerabilities, we have collected 60,577 real-world EOSIO contract samples directly from the EOSIO blockchain. Due to the absence of a centralized app market where contract code can be easily collected, we have been monitoring the blockchain activities to record any real-world transactions that are used to install contracts. Note that it is also possible to obtain deployed contract bytecode directly from the blockchain. However, two challenges arise: *(1)* EOS contracts are inherently upgradable, but only the most recent version is accessible on the blockchain, and *(2)* the lack of a batch-download API hinders the efficient retrieval of contracts. To completely and efficiently obtain contract code, we therefore extract WASM code from the transaction data. Nevertheless, the source code of these contracts is not available. The statistics of this dataset can be found in Appendix A-A.

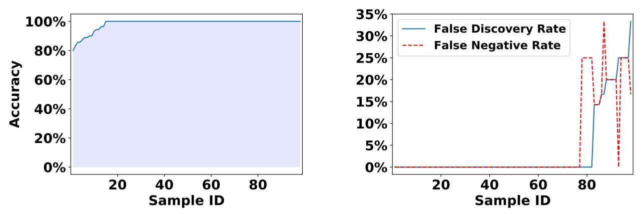
(c) We further aim to demonstrate that the state-of-the-art EOSIO contract vulnerability detector, EOSAFE [18], cannot precisely identify the GDVs, despite its effectiveness in detecting the "rollback" bugs. To this end, we have obtained the dataset used in EOSAFE which contains 715 contracts that are identified to be vulnerable to the "rollback" attacks.

System Settings. We have implemented a prototype system in 5,893 lines of Python code. Our experiments are conducted on a server equipped with Intel Xeon Gold 6330 CPU @ 2.00GHz, 256GB memory, and Ubuntu 20.04 LTS (64bit).

B. Accuracy of Static WASM Bytecode Analysis

To evaluate the accuracy of our static bytecode analysis, we apply VETEOS to the compiled WASM code of 98 EOSIO contracts. In the meantime, we also manually inspect the source code of these contracts to label the expected application entry points and dataflow paths. We then compare the automated detection results with the manually identified ground truths to derive the accuracy of VETEOS.

Entry Point Discovery. To assess the accuracy of our entry point detection, given a test sample, we randomly select points of interest (POI) in the code and use VETEOS to discover the entry points of each POI from WASM bytecode. We then consolidate the automatically discovered entry points for all the POIs, and manually find the total number of relevant entry points for the same set of POIs in the source code. The accuracy is finally calculated by dividing the number of discovered entry points by the actual total amount. Figure 12a represents the distribution of detection accuracy for entry points. The accuracy is calculated for each EOSIO contract program. The x-axis denotes the contract samples and the y-axis is the percentage of entry points that can be correctly identified. The results are sorted in the ascending order.



(a) Entry Point Discovery (b) Dataflow

Fig. 12: Accuracy of Static Analysis

The detection accuracy is on average 98.5%, and can reach 100% for 85.7% of the samples. The major reason that causes the inaccurate results is due to compiler optimization. For instance, in the `vigor.wasm`, while the action `strived()` is being called in the source code and thus is not considered to be an entry point, it is however not invoked by any functions in the bytecode. Instead, an optimized version of this function (i.e., `func117`) is generated and used in the WASM code. As a result, VETEOS will mistakenly consider this `strived()` action as an uncalled entry point. Notice that such an inaccuracy does not further affect our vulnerability detection because we did not miss the real entry point that makes a call to the optimized alternative.

Dataflow. To measure the accuracy of our dataflow analysis, we first locate multiple ground-truth dataflow paths in WASM bytecode. To do so, we manually pinpoint sources, sinks, and data paths in the source code. After instrumenting these statements and executing the code, we identify the instrumented bytecode instruction trace. This trace is then employed as the ground-truth for data paths. On average, for the 98 sample contracts, we labeled 4 paths, with each path comprising 397 WASM instructions. Then, starting from the sources of the paths, we utilize VETEOS to search for dataflow in the bytecode. Finally, we check how many bytecode instructions in the ground-truth paths are missed by our analysis (i.e., false negatives), and how many additional instructions are mistakenly included (i.e., false positives). Thus, we can compute the false negative rate `FNR = FN/(FN+TP)` and false discovery rate `FDR = FP/(FP+TP)` for each sample.

Figure 12b illustrates the result. The red dashed curve and blue curve represent the distributions of FNR and FDR, respectively, for the 98 samples. The average FNR is 4.43%, while the average FDR is 3.41%. For a large portion (79.6%) of contract samples, our dataflow analysis can achieve zero FDR and FNR. We further manually inspect the false negative and false positive cases to understand the root causes. Our study shows that both FNs and FPs are mainly caused by how we handle the memory address aliasing. For instance, in the case `salescon.wasm`, because our points-to analysis uses an overly strict rule – i.e., assessing the equality of base addresses and offsets individually, rather than determining if their combined sums match – we can actually miss certain data paths and cause incomplete results. In the meantime, in the cases such as `oracle.new.wasm` and `eosuber.wasm`, our false dataflows are caused by the classic challenge of over-tainting – when an aggregate data structure is copied while only a small portion within this structure is tainted, we conservatively consider the entire copy to be tainted.

TABLE II: Detecting Vulnerabilities by Steps

Factor	Semantics	# of Contracts
F1	`payToPlay` $\rightarrow$ `writeState` $\rightarrow$ `notify`	3,702/60,577

F2 & F1	`payToPlay` $\rightsquigarrow$ `(user_input)` $\rightsquigarrow$ `checkCondition`
	`createSecret` $\rightsquigarrow$ `(secret)` $\rightsquigarrow$ `checkCondition`
		3,394/3,702

F3 & F2 & F1	`writeState` $\rightsquigarrow$ `(global_state)` $\rightsquigarrow$ `readState`	2,086/3,394
F4 & F3 & F2 & F1	`checkCondition` $\rightarrow$ `writeState`	735/2,086

C. *Real-world Vulnerability Detection*

Overall Detection Results. We then apply VETEOS to real-world EOSIO contracts. Table II demonstrates how we capture the critical vulnerability factors in stages. First, our entry point detection and control-flow analysis identifies 3,702 samples out of the total 60,577 that satisfy the "rollback" requirement. Next, our dataflow analysis further confirms that 3,394 instances in the 3,702 contracts pass both a user input and an internally generated data item (possibly a secret) to a condition check, and 2,086 of them can actually leak internal states. Finally, by determining the control dependency between the condition check and the information leakage, we discover 735 samples that can actually be exploited by GHD attacks. A case study is presented in Appendix A-B.

We then manually verify the correctness of the result. We sequentially check the existence of the four factors: F1 Revertable – the initial payment (`apply()` function monitoring a notification from `eosio.token::transfer()`) and final notification (transferring tokens to a user or calling `require_recipient()`) exist in a single action or action sequence; F2 Unpredictable profitable – the user input (reachable from action interface) is compared with a contract property (a global state such as a table entry); F3/F4 Information leakage/Causal inference – the prior condition check guards a state update (e.g., change of table entry, calling `eosio.token::transfer()`) which is observable publicly.

False Positives. We randomly select 40 samples from the 735 cases and examine their WASM bytecode with manual efforts. Eventually, we have identified zero false positives – all the 40 contracts are confirmed to have the Groundhog Day vulnerabilities. We argue that the high precision can be attributed to two reasons: *(1)* we use strict rules to handle memory accesses and *(2)* the dataflow that exhibit in the Groundhog Day vulnerability pattern is not overly complex and usually does not involve sophisticated memory aliasing.

False Negatives. To assess whether VETEOS may miss any GDV instances, we run it on benchmark contracts. In particular, we have collected 36 samples identified by EOSAFE as having rollback issues. Out of these contracts, 18 have been reported by PeckShield [27], a leading Chinese blockchain security company, to have been attacked, while the remaining 18 have been manually verified by EOSAFE authors. Based upon the already discovered rollback problems, with EOSAFE authors' help, we further manually check if they have GDVs – i.e., whether these rollbackable transactions contain other factors including user controlled comparison, conditional state updates and exposed states. Eventually, we have confirmed that 18 samples contain all four essential factors of GDVs. In these 18 benchmark contracts, VETEOS successfully identify all the vulnerabilities without any false negatives.

Logic-level False Positives. In principle, our detection algorithm is susceptible to logic-level false positives. This is because our algorithm by design uses the disclosure of *global variables* to approximate the leakage of *game outcomes*. Nevertheless, the mere exposure of a global variable does not necessarily grant an attacker the essential knowledge. For instance, if there exists a state leakage only when a player loses, attackers remain uncertain whether their inputs can induce a winning scenario. Fundamentally, if the revealed global variables are irrelevant to critical game outcomes – for example, leaking a counter which is updated every time a game concludes but does not pertain to game results – they do not provide attackers with actionable information for launching a GHD attack. We admit that this is a fundamental limitation of our analysis, as it is a challenging task to infer the semantics of the identified global variables from opaque WASM bytecode, and we leave it to our future work. However, we did not observe such cases in our verified samples.

D. Financial Impact

The 735 identified contracts are deduplicated cases involving highly active EOSIO contracts, such as *Ge***OS*, *so***ys*, *ch***se* and *re***gm*, top apps on *eosauthority.com*. Of these, 10% generate over 2K transactions daily. For context, the average transaction count for the top 2000 EOSIO contracts stands at 1.3K. The total balance of the identified contracts currently amounts to 899K USD which can be directly affected. Moreover, because the GHD attacks target individual application instances (e.g, a lottery game, a number guessing game, etc.), any additional funds that are invested by participants *dynamically* will also be affected. For instance, a lottery game runs for one week with a daily transaction amount of 700K USD (on par with that of the top vulnerable contracts we have detected), the estimated affected balance added to that game instance would be 4.9M USD.

E. Disclosure

Identifying the developers of the detected contracts from the bytecode programs we gathered from transactions is notably challenging. This difficulty arises because the EOSIO ecosystem lacks a centralized knowledge base, similar to what Etherscan offers for Ethereum. Hence, we have to resort to limited symbol information such as EOSIO account names, Twitter accounts (e.g., *dr***on*, *eo***11*, *ch***dg*), Discord or Telegram groups (e.g., *ma***yz*, *fi***em*, *da***yp*, *z1***n1*) and cross-referenced them in GitHub (e.g., *eo***ps*, *il***ok*, *fi***em*) or Google (e.g., *ge***ol*, *pi***it*, *ch***se*, *be***io*, *ac***k1*, *il***ok*) to discover possible developers. We subsequently contact the identified developers to confirm ownership. Once verified, we share our findings with them.

We have reached out to 83 developers concerning the vulnerabilities, and the existence of their vulnerabilities has been confirmed. These vulnerabilities can also be verified in their published source code (e.g., *eo***ps*, *il***ok*, etc.). Among the responses, we have received feedback from the developers of *Ge***OS*, a leading EOS application. While they acknowledged the ownership of the contracts, they chose not to disclose specifics of their closed-source contracts. They claimed that their contracts, in their current states, are resilient to the GHD attacks, due to the external protection for their input interfaces, which prevents attackers from directly feeding inputs into the core app logic. However, we believe that if the fundamental vulnerability persists, it could potentially become exposed, for instance, during code refactoring.

We have reported our findings to CISA [9]. Particularly, we have introduced the formal definition of the Groundhog Day vulnerabilities, described our approach to automatically detecting this class of security bugs, and submitted the list of 735 vulnerable contract names we have identified. Per CISA's request, we have further provided the developer information of these contracts in order to assist them in confirming the vulnerabilities. In addition to CISA, we have attempted to present our results to other vulnerability disclosure programs. However, they do not accept EOSIO contract vulnerabilities due to the lack of vendor (i.e., EOSIO) collaboration.

F. Practicality of GHD Attacks

The efficacy of GHD attacks depends on *(1)* a notably high probability of triggering a winning condition and *(2)* an attacker's ability to gather necessary resources for sufficient attempts. Hence, we aim to comprehend how feasibly attackers can leverage this type of defects for financial advantage. To this end, we *(a)* investigate past occurrences of GHD attacks, *(b)* examine real-world contracts with GDVs to assess their likelihood of being exploited, *(c)* estimate attackers' capability via gauging the potential resources at their disposal, and *(d)* employ fuzzing on our identified GDV cases to quantitatively evaluate the cost-effectiveness of targeting such contracts.

(a) Existing GHD Attacks. Of the 18 benchmark contracts confirmed by EOSAFE authors and us to contain GDVs (used for our false negative test), several, including *dicecenter11*, *fairdogegame*, and *gamebetdices*, have already fallen victim to real-world attacks. EOSAFE authors note that these attacks have been inspected and documented by security experts at PeckShield [27]. Note that, PeckShield has already converted all their original intelligence reports to proprietary documents, making them inaccessible to the public. Nevertheless, we have discovered the PeckShield GitHub page which maintained a list of previously identified attacks [26] and news articles (in Chinese) that reported PeckShield's findings about the attacks against *dicecenter11*, *fairdogegame*, *gamebetdices* contracts [3]. In addition, using the reported names of the victim contracts and the dates of attacks, we cross-referenced the data with *bloks.io* (a major EOS block explorer). This has led us to identify unusually high-volume (potentially malicious) transactions associated with these contracts [3].

(b) Analysis of Winning Conditions. We further study the source code we have discovered for two real-world number-guessing game contracts, *EOSBet Casino* [5] and *EOS.Win* [13], that are susceptible to GHD attacks. Our investigation indicates that merely $\sim$90 attempts are needed to execute the GHD attacks against these contracts. We present our annotated contract code for these contracts in Appendix A-C. In general, these games generate secret numbers between the range *[1,100]* or *[0,99]*. Though designed to be random, the actual randomness is compromised because of an improperly used constant seed. Subsequently, these apps accept user inputs from either *[2,96]* or *[2,97]*. If a user input is greater or less than the secret number, they win. Consequently, in most scenarios (when the secret falls between *[3,95]*), attackers would require a maximum of $\sim$90 tries to meet the winning criteria. These real-world examples demonstrate that, although a wide range of secret values can make attacks more challenging in theory, in practice, applications may still utilize a limited secret range – this leaves them vulnerable to GHD attacks.

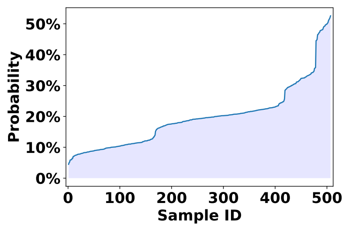


Fig. 13: Attack Success Rate through Fuzzing

(c) Estimation of Attackers' Capability. According to the statistics provided by EOS Authority [11], an average transaction consumes 286.25 μs of CPU time and 165.6 Byte NET, and the average tokens staked by each EOS holder amount to 977 USD (equivalent to 8,724s CPU + 16,631 MB NET). As a result, an average user could potentially execute a contract up to 30M times. This makes launching attacks against GDV contracts, such as the aforementioned ones, practically viable. Besides, any investments made by GHD attackers are, in essence, "reimbursable" owing to the rollback issue and the fact that the tokens staked for CPU/NET can be reclaimed. This means that even if attackers lack the necessary resources for a sufficient number of attempts, thereby inhibiting their capacity to deduce the winning conditions, they are not financially compromised. These cost-free retries grant them adequate opportunities to discover and capitalize on vulnerable contracts (which are more likely to be exploited).

(d) Dynamic Verification. To show the cost-effectiveness of GHD attacks, we employed WASAI [7], an open-source fuzzing tool, to assess the exploitability of the 735 identified GDV samples. The inputs of WASAI consist of a contract's WASM program and its Application Binary Interface (ABI). Utilizing the specific ABI information enables the fuzzer to bypass ill-structured inputs, which a logical attacker would unlikely provide. It is noteworthy that identifying the correct ABI for a target EOS function is non-trivial. Although EOSIO blockchain explorers such as EOS Authority publish ABI information, they only provide the most recent function interfaces. Yet the vulnerable code we have detected may exist solely in particular older versions of these contracts, requiring specific past versions of ABIs for execution. Hence, we search for ABI data from historical EOS transaction logs – where every contract deployment is recorded – based upon the transaction timestamps and indices. Finally, we manage to identify the ABIs of vulnerable functions for 507 out of the 735 samples.

To quantify the probability that an attacker can successfully activate the winning conditions within the vulnerable functions, we instrument both the success and failure branches for every GDV-related condition check that VETEOS has identified. Then, we run each of the 507 samples 500 times using random initial input values, tallying both successful and failed attempts. Thus, the proportion of successful trials, depicted in Figure 13, serves as the estimated likelihood of successfully executing a GHD attack. As you can see, the average success rate is 19%, while the lowest is still around 5%. This indicates that the discovered vulnerabilities can be practically exploited with a reasonable amount of attempts.

TABLE III: Identified Vulnerability Factors

Factor	EOSAFE	VETEOS
F1	715/715	715/715
F2 & F1	715/715	563/715
F3 & F2 & F1	NA	195/715
F4 & F3 & F2 & F1	NA	144/715

G. *Limitation of* EOSAFE

We then investigate whether and how EOSAFE [18] falls short in detecting GDVs. Note that EOSAFE is not a tool specifically designed to address this information flow-based issue but rather a symbolic execution engine that can identify "rollbackable" control flows (F1). Nevertheless, because EOSAFE adopts simple heuristics – such as searching for modulo instructions (i.e., `rem`) that can be used to generate random numbers (i.e., secrets), it has actually been used by their authors to detect GDVs. However, this fundamental lack of formal vulnerability modeling may potentially lead to significant false positives. To verify this, we have obtained 715 samples from the authors of EOSAFE. These contracts have been flagged by EOSAFE as being vulnerable to "rollback" attacks but have not been verified manually by the authors.

To determine the existence of the GDVs in these contracts, we apply VETEOS to their WASM code to search for the four enabling factors. Table III depicts the results. While the prior work can successfully recognize all the rollbackable actions (*F1*), its detection of *F2 & F1*, meaning rollbackable actions that allow users to make profits in an unpredictable fashion, is not precise because it does not check whether an identified random number is actually compared with user inputs. In contrast, VETEOS uses a more accurate rule for *F2 & F1* that checks if user inputs are used in an internal comparison, and thus discovers less cases satisfying the condition. Besides, EOSAFE does not further consider the other factors (*F3* and *F4*), as it does not detect leakage flow or build the dependency between user inputs and the leakage flow. VETEOS, by additionally investigating these two factors, can precisely remove 79.8% of the cases which cannot be exploited to mount GHD attacks.

H. *Runtime Performance*

We finally assess the runtime performance of VETEOS using the 60K real-world contracts. Overall, our detection is fast due to its nature of static program analysis. On average, it takes 3.46 seconds to process a bytecode program. Particularly, our callgraph analysis and entry point detection cost 2.6 seconds per app, and the dataflow analysis takes 0.86 seconds.

VII. MITIGATION

Groundhog Day vulnerabilities are realistic and serious security problems in the EOS ecosystem. Conceptually, this kind of issues may even affect smart contracts on other blockchain platforms, as discussed in Appendix A-D. To mitigate such problems, we propose two strategies *(1)* separating funds transfers from core game logic and *(2)* hiding critical global contract states. The first defense strategy aims to disrupt the *F1 Revertable* factor. Essentially, a GDV exists because an adversary can revert an entire transaction, which encompasses not only checking a user input against the secret, updating global states, but also informing the user of the outcome. Hence, reversing the final notification can unnecessarily also revert the prior steps implementing the core game logic. To avoid this, we can split this sequence of operations into multiple separate transactions. Thus, the F1 factor no longer holds as the "rollback" issue merely affects the notification process. In contrast, the transactions related to the central application logic have already completed and cannot be undone. The second possible solution targets obscuring critical global states that may leak game statuses (i.e., *F3 Information leakage*). Specifically, we propose to restrict public accesses to global states such as account balances stored in tables managed by contracts. Since game outcome are communicated to end users via notifications, blocking access to their balances does not notably affect usability. Meanwhile, as attackers cannot prematurely ascertain the game's result, their ability to secure unwarranted profits predictably is negated.

VIII. Related Work

Security of EOSIO Contracts. Prior work has developed tools to discover security problems in EOSIO WASM bytecode. EOSafe [18] proposes the first static WASM bytecode analysis framework. WANA [36] introduces a cross-platform vulnerability detection tool based on the symbolic execution. EOSFuzzer [19] develops a general black-box fuzzing framework to detect EOSIO contract vulnerabilities. WASAI [7] implements a new concolic fuzzer for uncovering vulnerabilities in WASM contract programs.In contrast, VetEOS is the first work in this domain that provides the capability of conducting static dataflow analysis and detecting the novel GDVs.

Security Vetting of Smart Contracts. Many efforts [24], [37], [35], [20], [21], [16], [31], [32], [28], [34], [15] have been made to automatically verify smart contract code to detect security risks. Existing tools detect both syntax-based errors [24], [21], [16], [31], [15] and semantic-level defects [35], [20], [28], [34], [33]. In comparison, VetEOS detects security bugs in a completely different type of contracts, and thus must address the unique high-level programming paradigm and low-level implementation mechanisms used in EOSIO WASM code.

Static Analysis of Event-Driven Programs. Static analysis tools have been built to analyze event-driven programs. Due to the special mechanisms for triggering such applications, these analyzers must particularly identify program entry points. Efforts have been made to address this challenge in different application domains (e.g., Android [23], [42] or industrial controllers [41]). VetEOS follows the same idea but models such entry points in a more strict manner.

IX. Conclusion

We propose VetEOS, a static vetting tool for the *Groundhog Day vulnerabilities* in EOSIO contracts. VetEOS formally defines this unique vulnerability as a control and data dependency problem, addresses multiple distinct challenges for analyzing EOSIO WASM programs, and has detected 735 new vulnerabilities in the wild.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive comments. We thank Weimin Chen for providing thoughtful insights during the fuzzing experiments. This work was supported in part by NSF OAC-2115167, NSF DGE-2041960, DARPA HR00112120009 Cooperative Agreement, a Cisco gift and a USHE Deep Technology Initiative Grant. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agency.

References

[1] "Octopus: Security Analysis tool for Blockchain Smart Contracts," https://github.com/FuzzingLabs/octopus, 2023.

[2] "The Ethereum Blockchain Explorer," https://etherscan.io/, 2023.

[3] "VetEOS," https://github.com/HKJL10201/VetEOS, 2023.

[4] 101 Blockchains, "Delegated Proof Of Stake (DPoS) – Explained," https://101blockchains.com/delegated-proof-of-stake-dpos/, 2023.

[5] BestBitcoinCasino, "EOSBet Casino," https://www.bestbitcoincasino.com/review/eosbet-casino/, 2023.

[6] BSTX, "BSTX is building the first blockchain-integrated national securities exchange," https://bstx.com/, 2023.

[7] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "WASAI: Uncovering vulnerabilities in wasm smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022, 2022.

[8] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Static Analysis*, 2003.

[9] CISA, "Coordinated Vulnerability Disclosure Process," https://www.cisa.gov/coordinated-vulnerability-disclosure-process, 2023.

[10] CoinMarketCrab, "EOS," https://coinmarketcap.com/currencies/eos/, 2023.

[11] EOSAuthority, "REX," https://eosauthority.com/rex/statistics?network=eos, 2023.

[12] EOSIO, "Staking on EOSIO-based blockchains," https://developers.eos.io/manuals/eosio.contracts/latest/key-concepts/stake, 2023.

[13] EOS.Win, "EOS.Win – purely on-chain decentralized platform," https://github.com/eoswindev/contract/blob/master/dice/dice.cpp#L787, 2023.

[14] Ethereum, "VERIFYING SMART CONTRACTS," https://ethereum.org/en/developers/docs/smart-contracts/verifying/, 2023.

[15] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[16] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," in *Procceedings of The 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2018)*, 2018.

[17] N. He, W. Su, Z. Yu, X. Liu, F. Zhao, H. Wang, X. Luo, G. Tyson, L. Wu, and Y. Guo, "Understanding the evolution of blockchain ecosystems: A longitudinal measurement study of bitcoin, ethereum, and EOSIO," https://arxiv.org/abs/2110.07534, 2021.

[18] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "EOSAFE: Security analysis of EOSIO smart contracts," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[19] Y. Huang, B. Jiang, and W. K. Chan, "EOSFuzzer: Fuzzing EOSIO smart contracts for vulnerability detection," in *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, ser. Internetware '20, 2021.

[20] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proceedings of the 2018 Network and Distributed System Security Symposium*, 2018.

[21] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[22] S. Lee, D. Kim, D. Kim, S. Son, and Y. Kim, "Who spent my EOS? on the (In)Security of resource management of EOS.IO," in *13th USENIX Workshop on Offensive Technologies (WOOT'19)*, 2019.

[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, 2012.

[24] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[25] M. Mohri and M.-J. Nederhof, "Regular approximation of context-free grammars through transformation," in *Robustness in Language and Speech Technology*, 2001.

[26] PeckShield, "PeckShield," https://github.com/peckshield/EOS/tree/master/known_dapp_attacks, 2023.

[27] ——, "PeckShield Industry Leading Blockchain Security Company," https://peckshield.com/, 2023.

[28] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[29] T. PROEBSTING, "Writing a Sealed-Bid Auction Contract," https://programtheblockchain.com/posts/2018/03/27/writing-a-sealed-bid-auction-contract/, 2023.

[30] L. Quan, L. Wu, and H. Wang, "EVulHunter: Detecting fake transfer vulnerabilities for EOSIO's smart contracts at webassembly-level," http://arxiv.org/abs/1906.10362, 2019.

[31] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proceedings of the 2019 Network and Distributed System Security Symposium*, 2019.

[32] E. Shishkin, "Debugging smart contract's business logic using symbolic model checking," *Programming and Computer Software*, vol. 45, no. 8, pp. 590–599, 2019.

[33] S. So, S. Hong, and H. Oh, "SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language Model-Guided symbolic execution," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[34] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[35] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018.

[36] D. Wang, B. Jiang, and W. K. Chan, "WANA: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection," https://arxiv.org/abs/2007.15510, 2020.

[37] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, and I. Naseer, "Formal specification and verification of smart contracts for azure blockchain," http://arxiv.org/abs/1812.08829, 2019.

[38] Wikipedia, "Double auction," https://en.wikipedia.org/wiki/Double_auction, 2023.

[39] ——, "First-price sealed-bid auction," https://en.wikipedia.org/wiki/First-price_sealed-bid_auction, 2023.

[40] ——, "Proof of work," https://en.wiki pedia.org/wiki/Proof_of_work, 2023.

[41] M. Zhang, C.-Y. Chen, B.-C. Kao, Y. Qamsane, Y. Shao, Y. Lin, E. Shi, S. Mohan, K. Barton, J. Moyne, and Z. M. Mao, "Towards automated safety vetting of plc code in real-world plants," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[42] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, 2014.

Appendix A
Additional Data and Discussion

A. *Statistics of the Evaluation Dataset.*

We survey the 60,577 samples to understand their nature. Firstly, these contracts contain fairly large WASM bytecode programs, with an average size being 80KB and the largest app, eo***iv, being 522KB. Secondly, these contracts have been recently deployed, from 2018 to 2023, and new apps have been constantly introduced. In general, 15K new contracts are being installed every year. Thirdly, we further expect to categorize what activities these apps may perform. Because the source code or any textual description of these applications is unavailable, we alternatively utilize the string constants stored in these contracts to estimate their "semantics". Figure 14 shows the word cloud we can build from the string literals extracted from these WASM code. This indicates that our collected samples commonly perform EOS token transfers and may include different types of applications such as game, lottery, token exchange market, decentralized finance, etc.

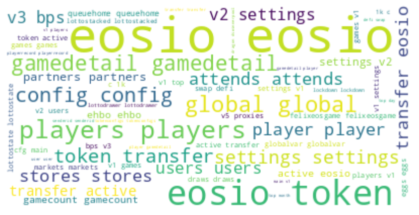


Fig. 14: Word Cloud of the String Literals

B. *Case Study for Real-world GDV Detection.*

Figure 15 illustrates how we capture the Groundhog Day vulnerability in a real-world EOSIO contract, *EOSBet Casino* [5]. This casino app enables users to wager on a specific number, and the players win if their chosen number matches a randomly generated secret number.

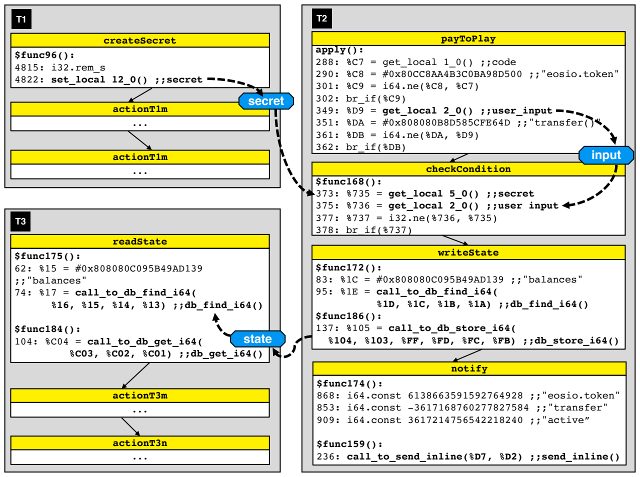
VetEOS has discovered the four enabling factors of the Groundhog Day vulnerability in this WASM program. More concretely, *(F1) Revertable:* VetEOS identifies the sequence of activities, from "payToPlay" (`apply()` that handles token transfer notification) to "notify" (`send_inline()`), that can be executed in the same transaction and therefore can be reverted as an entirety. *(F2) Unpredictably profitable:* our analysis also discovers that the outcome of the game depends on a successful comparison between a user input (`get_local 2_0()`) and a secret (`get_local 5_0()`). *(F3) Information leakage:* we further uncover the information flow from a database table write (`db_store_i64()`) to a table read (`db_get_i64()`). Finally, *(F4) Causal inference:* our detector establishes the control dependency between the condition check and the account balance update (`db_find_i64(,balances,,)` and `db_store_i64()`)

This example, again, demonstrates that, in order to accurately detect the Groundhog Day vulnerabilities hidden in EOSIO WASM code, a static analyzer must be able to *(1)* accurately identify the application entry point so as to capture an entire activity sequence and *(2)* precisely discover the dataflow across multiple actions so as to build necessary causal relations and leakage flows.

C. *Annotated Source Code for EOSBet Casino and EOS.Win*

Figure 16 and Figure 17 illustrate the source code of *EOSBet Casino* and *EOS.Win*, respectively. These two contracts implement a number guessing game, where the range of the secret number is limited. We annotate the original source code with comments to describe its application logic and explain the scope of the success condition.

As shown in Figure 16, the core workflow of *EOSBet Casino* begins with the extraction of transfer information using `unpack_action_data()` (LN4). The user input is obtained from the *memo* field of the transfer data, then parsed and



```
$func96():
4815: i32.rem_s
4822: set_local 12_0() ;;secret
```

```
apply():
288: %C7 = get_local 1_0() ;;code
290: %C8 = #0x80CC8AA4B3C0BA98D500 ;;"eosio.token"
301: %C9 = i64.ne(%C8, %C7)
302: br_if(%C9)
349: %D9 = get_local 2_0() ;;user_input
351: %DA = #0x808080B8D585CFE64D ;;"transfer()"
361: %DB = i64.ne(%DA, %D9)
362: br_if(%DB)
```

```
$func168():
373: %735 = get_local 5_0() ;;secret
375: %736 = get_local 2_0() ;;user input
377: %737 = i32.ne(%736, %735)
378: br_if(%737)
```

```
$func172():
83: %1C = #0x808080C095B49AD139 ;;"balances"
95: %1E = call_to_db_find_i64(
          %1D, %1C, %1B, %1A) ;;db_find_i64()
$func186():
137: %105 = call_to_db_store_i64(
  %104, %103, %FF, %FD, %FC, %FB) ;;db_store_i64()
```

```
$func174():
868: i64.const 6138663591592764928 ;;"eosio.token"
853: i64.const -3617168760277827584 ;;"transfer"
909: i64.const 3617214756542218240 ;;"active”

$func159():
236: call_to_send_inline(%D7, %D2) ;;send_inline()
```

```
$func175():
62: %15 = #0x808080C095B49AD139
;;"balances"
74: %17 = call_to_db_find_i64(
      %16, %15, %14, %13) ;;db_find_i64()

$func184():
104: %C04 = call_to_db_get_i64(
         %C03, %C02, %C01) ;;db_get_i64()
```

Fig. 15: Groundhog Day Vulnerability in *EOSBet Casino*

```
1  class EOSBetDice : public eosio::contract {
2      void transfer(uint64_t sender, uint64_t receiver) {
3          // Get transfer data from unpacked action data
4          auto transfer_data = unpack_action_data<
       st_transfer>();
5          const std::size_t first_break = transfer_data.memo
       .find("-");
6          // Get user input (guessing number) from transfer
       data
7          roll_str = transfer_data.memo.substr(0,
       first_break);
8          const uint64_t roll_under = std::stoull(roll_str,
       0, 10);
9          // Restrict user input in [2,96]
10         eosio_assert( roll_under >= 2 && roll_under <= 96,
        "Roll must be >= 2, <= 96.");
11         // Store the betting information in the table
12         activebets.emplace(_self, [&](auto& bet){
13             bet.id = bet_id;
14             bet.bettor = transfer_data.from;
15             bet.referral = referral;
16             bet.bet_amt = your_bet_amount;
17             bet.roll_under = roll_under;
18             bet.seed = seed_hash;
19             bet.bet_time = time_point_sec(now());
20         });
21     }
22     void resolvebet(const uint64_t bet_id, signature sig)
        {
23         // Read the betting information from the table
24         auto activebets_itr = activebets.find( bet_id );
25         eosio_assert(activebets_itr != activebets.end(), "
        Bet doesn't exist");
26         // Create secret number in [1,100]
27         const uint64_t random_roll = ((random_num_hash.
       hash[0] + random_num_hash.hash[1] + random_num_hash.
       hash[2] + random_num_hash.hash[3] + random_num_hash.
       hash[4] + random_num_hash.hash[5] + random_num_hash.
       hash[6] + random_num_hash.hash[7]) % 100) + 1;
28         // Compare secret (random_roll) with user input (
       roll_under), checking (secret < user input)
29         if(random_roll < activebets_itr->roll_under){
30             payout = (activebets_itr->bet_amt *
       get_payout_mult_times10000(activebets_itr->roll_under
       , edge)) / 10000;
31         }
32         // Update the global state
33         increment_game_stats(activebets_itr->bet_amt,
       payout);
34         // Send the rewards to the player
35         action(
36             permission_level{_self, N(active)},
37             N(eosio.token),
38             N(transfer),
39             std::make_tuple(
40                 _self,
41                 activebets_itr->bettor,
42                 asset(payout, symbol_type(S(4, EOS))),
43                 std::string("Bet id: ") + std::to_string(
       bet_id) + std::string(" -- Winner! Play: dice.eosbet.
       io")
44             )
45         ).send();
46     }
47 };
```

Fig. 16: Annotated Source Code of *EOSBet Casino*

checked to ensure it falls within the range *[2,96]* (LN7, LN10). Subsequently, this user input, along with other betting details, is stored in the *activebets* table (LN12-20). When a bet needs to be resolved, the `resolvebet()` function is called, which retrieves the betting information from the *activebets* table (LN24). A secret number in the range *[1,100]* is randomly generated (LN27), and a comparison is made to check if the user input is greater than the secret (LN29). The global game state, stored in the *activebets* table, is then updated with the result (LN33). Finally, the rewards are sent to the player using an inline action (LN35-45).

Similarly, in EOS.Win (Figure 17), transfer information is acquired at LN4, and a user input is extracted from the transfer *memo* at LN9. This user input is restricted to the range *[2,97]* as shown in LN11. Betting details, including the user input, are stored in a table indicated by *r_out.actions* (LN13). The `resolved()` function receives betting information as parameters (LN17) and generates a random secret number within the *[0,99]* range (LN19), then compares the user input with the secret based upon the roll type (small or big) (LN21). If the player wins, the global game state stored in the *trades* table is updated (LN26-28), and rewards are sent to the player through an inline action (LN31).

D. Generality of Groundhog Day Vulnerability

This study concentrates specifically on the Groundhog Day vulnerability within EOSIO contracts. However, this type of vulnerability could also potentially exist in other forms of smart contracts, such as Ethereum contracts, provided that the four enabling factors are present.

In fact, three factors – *F2 Unpredictably Profitable*, *F3 Information Leakage* and *F4 Causal Inference* – are defined at the logic level and can, therefore, appear in any financial applications regardless of the underlying programming language features. In contrast, the first factor, *F1 Revertable*, is especially achievable in EOSIO smart contracts due to the specific “rollback” attacks. Since users are able to receive notification when a transaction is near completion, it opens the door for adversaries to consistently revert the entire transaction.

Although other contract languages such as Ethereum’s Solidity may not present such a direct attack surface, attackers could still intentionally revert specific transactions. It is actually not uncommon for a Solidity function to explicitly call the `revert()` API under certain conditions. For instance, a bidding function must verify whether an auction has expired or if the bid is legitimate, and will abort the entire transaction if the condition is not satisfied. Since source code of Solidity contracts is often publicly available on, for example, Etherscan [2] – as this allows third parties and contract users to verify the equivalence between source code and deployed bytecode [14] – attackers can gain insights into contract implementations, especially about transaction reversibility. Particularly, if the

```
class dice : public eosio::contract {
    void transfer(account_name from, account_name to, eosio::asset quantity, string memo) {
        // Get transfer data from unpacked action data
        eosio::currency::transfer t = {from, to, quantity, memo};
        vector<string> pieces;
        boost::split(pieces, t.memo, boost::is_any_of(","));
        // Get user input (guessing number) from transfer data
        uint8_t roll_type = atoi( pieces[0].c_str() );
        uint64_t roll_border = atoi( pieces[1].c_str() );
        // Restrict user input in [2,97]
        eosio_assert(roll_border >= ROLL_BORDER_MIN && roll_border <= ROLL_BORDER_MAX, "Bet border must between 2 to 97");
        // Store the betting information in the table
        eosio::transaction r_out;
        auto t_data = make_tuple(t.from, t.quantity, roll_type, roll_border, inviter);
        r_out.actions.emplace_back(eosio::permission_level{_self, N(active)}, _self, N(start), t_data);
    }
    void resolved(account_name bettor, eosio::asset bet_asset, uint8_t roll_type, uint64_t roll_border, account_name inviter) {
        // Create secret number in [0,99]
        uint64_t roll_value = get_random(BET_MAX_NUM);
        // Compare secret (roll_value) with user input (roll_border), checking user input is less or greater than secret
        bool is_win = (roll_type == ROLL_TYPE_SMALL && roll_value < roll_border) || (roll_type == ROLL_TYPE_BIG && roll_value > roll_border);
        if ( is_win )
        {
            // Update the global state
            int64_t reward_amt = get_bet_reward(roll_type, roll_border, bet_asset.amount);
            _trades.modify(trade_iter, 0, [&](auto& a) {
            a.out += reward_amt;
            });
        }
        // Send the rewards to the player
        INLINE_ACTION_SENDER(eosio::token, transfer)(lucky_trade_iter->contract, {_self, N(active)}, {_self, bettor, lucky_iter->reward, string(str)} );
    }
};
```

Fig. 17: Annotated Source Code of *EOS.Win*

condition check (and hence potential revert position) is mistakenly placed at a later point in a function, attackers could exploit this vulnerability, crafting specific inputs to deliberately revert the transaction while monitoring leaked state updates from a side channel. Admittedly, while feasible in theory, practical factors must also be taken into account. For instance, the cost of code execution (e.g., gas in Ethereum) may affect the financial gain of such attacks. However, adversaries that carefully consider the trade-off between cost and benefit may still make such attacks possible. In addition, attackers can identify other types of smart contracts that do not require any cost for execution.

Consequently, our current work is an exploration of this critical vulnerability, using EOSIO contracts as a case study. However, it is worth noting that demonstrating a general problem and its detection method within a specific programming language context is non-trivial, and requires to devise special mechanisms to handle distinct challenges originating from unique high-level programming paradigm and low-level implementation, such as memory modeling. The major contribution of this work lies precisely in this aspect. As a result, despite the fact that Groundhog Day vulnerability is a general problem in financial applications, we limit our discussion to its manifestation in EOSIO smart contracts at this point, leaving the investigation of this issue in other domains for future work.

Appendix B
Artifact Appendix

A. *Description & Requirements*

1) How to access:

- Zenodo: https://doi.org/10.5281/zenodo.10158696.
- GitHub: https://github.com/HKJL10201/VetEOS.

2) Hardware dependencies:

- Processor: Any 64-bit processor, such as Intel Core Processor Series.
- Memory: 4 GB RAM, recommend 8 GB or higher.
- Storage: 1 GB or higher available space.

3) Software dependencies:

- Operating System: macOS or Linux, recommend Ubuntu 16.04 or higher.
- Software: Python 3.7 or 3.8.
- Python Dependencies:
  - wasm
  - graphviz
  - timeout-decorator

4) Benchmarks: None.

B. *Artifact Installation & Configuration*

1) Install Python 3.7 or 3.8.
2) Download VetEOS from Zenodo or GitHub.
3) Install the dependencies using the script in VetEOS: `python3 install_dependencies.py`. Note that the script uses `pip` to install `graphviz`, if `pip` fails, please try `apt-get`: `sudo apt-get install graphviz`. For macOS users, the dependency `graphviz` needs to be installed manually.

C. *Major Claims*

- (C1): VetEOS has detected 735 new vulnerabilities in the wild. This is proven by the experiment (E2) whose results are reported in Table II.
- (C2): VetEOS achieves average 98.5% entry point detection accuracy. This is proven by the experiments (E4) whose results are illustrated in Figure 12a.
- (C3): VetEOS achieves average 4.43% False Negative Rate and 3.41% False Discovery Rate in dataflow analysis. This is proven by the experiments (E5) whose results are illustrated in Figure 12b.

D. *Evaluation*

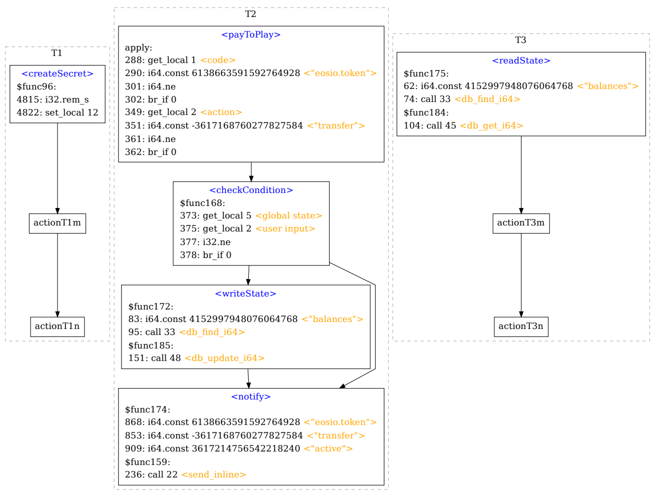
1) Experiment (E1): [GDV Analysis Test] [1 human-minute + 1 compute-minute]: Test the GDV analysis functionality of VetEOS.

[Preparation] Ensure that the working directory is the root directory of VetEOS project. Execute command `python3 tests/test_dependencies.py` to ensure that all dependencies are installed and available.

[Execution] `bash tests/test_GDV.sh`.

[Results] Expected printed output:

```
...
Total number of files analyzed: 24
Detected Groundhog Day Vulnerabilities: 24
Results are stored in ./results/
```



```
<createSecret>
$func96:
4815: i32.rem_s
4822: set_local 12

<payToPlay>
apply:
288: get_local 1 <code>
290: i64.const 6138663591592764928 <"eosio.token">
301: i64.ne
302: br_if 0
349: get_local 2 <action>
351: i64.const -3617168760277827584 <"transfer">
361: i64.ne
362: br_if 0

<checkCondition>
$func168:
373: get_local 5 <global state>
375: get_local 2 <user input>
377: i32.ne
378: br_if 0

<writeState>
$func172:
83: i64.const 4152997948076064768 <"balances">
95: call 33 <db_find_i64>
$func185:
151: call 48 <db_update_i64>

<notify>
$func174:
868: i64.const 6138663591592764928 <"eosio.token">
853: i64.const -3617168760277827584 <"transfer">
909: i64.const 3617214756542218240 <"active">
$func159:
236: call 22 <send_inline>

<readState>
$func175:
62: i64.const 4152997948076064768 <"balances">
74: call 33 <db_find_i64>
$func184:
104: call 45 <db_get_i64>
```

Fig. 18: Analysis Summary Graph Example

The logs and generated analysis summary graphs will be stored in `./results/`. An example of detailed analysis log can be found at: `./results/example.log`. Figure 18 shows an example of generated analysis summary graph, which also can be found at: `./results/example.pdf`.

2) Experiment (E2): [GDV Detection Test] [1 human-minute + 10 compute-minutes]: Run GDV detection test on 735 vulnerable samples.

[Preparation] Ensure that the working directory is the root directory of VetEOS project.

[Execution] `bash tests/test_GDV_all.sh`.

[Results] Expected printed output:

```
...
Total number of files analyzed: 735
Detected Groundhog Day Vulnerabilities: 735
```

3) Experiment (E3): [Dataflow Analysis Case Study] [1 human-minute + 1 compute-minute]: Test the dataflow analysis functionality of VetEOS on binary EOSIO smart contracts.

[Preparation] Ensure that the working directory is the root directory of VetEOS project.

[Execution] `bash tests/test_dataflow.sh`.

[Results] The script will trigger a process of automatic dataflow tracking testing through the VetEOS terminal. An example output of dataflow test can be found at: `./results/example_output_dataflow_test.log`.

4) Experiment (E4): [Entry Point Detection Test] [1 human-minute + 1 compute-minute]: Test the accuracy of entry point detection.

[Preparation] Ensure that the working directory is the root directory of VetEOS project.

[Execution] `python3 tests/tests.py entrypoint`.

[Results] Expected printed output:

```
...
samples:98
error:0
```

The test logs will be printed directly to the terminal. Note that accuracy tests requires manual verification against the source code. The source code of the tested samples can be found in `./samples/sourcecode.tar.gz`.

5) Experiment (E5): [Dataflow Accuracy Test] [1 human-minute + 1 compute-minute]: Test the accuracy of dataflow analysis.

[Preparation] Ensure that the working directory is the root directory of VETEOS project.

[Execution] `python3 tests/tests.py dataflow`.

[Results] Expected printed output:

```
...
dataflow false positive test:
test cases:537
error:0
dataflow false negative test:
test cases:537
error:0
```

The test logs will be printed directly to the terminal. Note that accuracy tests requires manual verification against the source code. The source code of the tested samples can be found in `./samples/sourcecode.tar.gz`.

E. Customization

To manually perform the GDV analysis functionality, run command: `python3 main.py -f <filepath> -g -d`, where the flag `-g` enables the analysis graph generation, and flag `-d` enables dumping the analysis logs.

For example, if need to dump the log files to directory `./results/` during experiment (E2), modify the line 24 of file `./tests/test_GDV_all.sh` to be `output=$(python3 main.py -f "$file" -g -d)`.

F. Notes

VETEOS is highly scalable, it includes an integrated terminal for visualizing instructions, accessible through the command `python3 main.py -v`. This feature enhances the intuitiveness and user-friendliness of dataflow analysis and allows for potential expansion of additional analysis of WASM bytecode capabilities in the future.