To play, call the play() method with the guessed number (1-20). If you're smart enough, you'll win in every round. To witness scam, call the scam() method with the guessed number (1-20). Your money will disappear every round. To load the honey pot, call deposit, or initialize the contract with the desired amount. Bet price: 0.1 ether.

Figure 12: Developer's Description

There is an Uninitialised struct problem.

Figure 13: Security Report

The function calculates a value using a timestamp, and then if the value is equal to a user input value, the function transfers the balance of the contract to user.

Figure 14: *Tx2TXT* Description

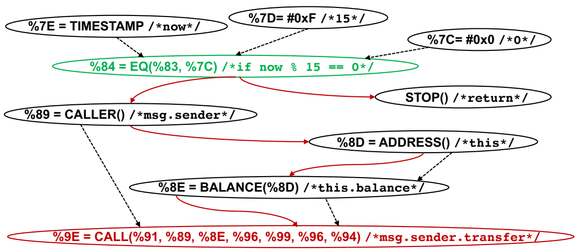

Figure 15: Example of Added Condition

A API MODELS

Table A1 depicts semantic models we have built for Solidity and ERC APIs. This is a partial table and we also support ERC-1155, ERC-721 and ERC-4626.

B EXAMPLE DESCRIPTIONS

Figure 12, Figure 13 and Figure 14 illustrate the textual description for the CryptoRoulette gambling game [19], from the developer, SECURIFY+HONEYBADGER and *Tx2TXT*, respectively. CryptoRoulette determines the winner of a game using block timestamp that can be controlled by attackers and therefore can cause fairness issues. In this case, *Tx2TXT* can correctly capture this problematic condition check and concisely present its consequence (i.e., unfair funds transfer) in natural language. In contrast, the developer's description and the security report are either security-insensitive or overly abstract, and thus may not be easily comprehensible to average users.

C CASE STUDY

Case Study. Figure 15 illustrates an example FTG where a critical condition is being added after node classification. Particularly, this contract [18] implements a gambling game [19], where each participant must pay to play. If the timestamp of a payment is the multiple of 15, the corresponding player wins and thus is rewarded with the entire balance of this contract. Due to the timestamp dependency problem, this game is not fair as the winner is determined by the block timestamp which can be manipulated by a miner. Our ML model correctly discovers this important condition check based upon the critical timestamp API, as well as the unbalanced branches – when the condition is not met, the function immediately returns (i.e., `STOP()`). In this case, we introduce only one condition node

Algorithm 2 Description Generation

```
procedure DescGen(FTG, M)
    DESC ← ∅
    FTG_ctrl ← GetAllControlTransfers(FTG)
    for ∀p ∈ FTG_ctrl do
        desc ← null
        for ∀node ∈ p do
            if GetAttr(node) == "condition" then
                {subj, obj} ← Concretize(DEF_node, M)
                CondMod ← RealizeSentence({subj, "be equal to", obj})
                desc ← Aggregate(desc, CondMod)
            else if GetAttr(node) == "transfer call" or "NCDO" then
                {vb, obj, mod} ← Concretize(DEF_node, USE_node, M)
                sentence ← RealizeSentence({"function", vb, obj, mod})
                desc ← Aggregate(desc, sentence)
            end if
        end for
        DESC ← DESC ∪ desc
    end for
    return DESC
end procedure
```

(green) into the original graph, while two other conditions that check payment amount and game availability are not selected.

D GCN DETAILS

Given an ACFG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with N nodes $v_i \in \mathcal{V}$, edges $(v_i, v_j) \in \mathcal{E}$, an adjacency matrix $A \in \mathbb{R}^{N\times N}$, a degree matrix $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ and a matrix X of node feature vectors X_i, a GCN model $Z = f(X, A)$ maps the inputs X and A to an output vector Z, which indicates the probability of each vertex being security-sensitive or not. Formally, a multi-layer GCN follows this layer-wise propagation rule:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{2}$$

where $\tilde{A} = A+I_N$ is the adjacency matrix with added self-connections, denoted as the identity matrix I_N. $W^{(l)}$ is a layer-specific trainable weight matrix. $\sigma(\cdot)$ represents an activation function, such as the $\mathbf{ReLU}(\cdot)$ [35]. $H^{(l)} \in \mathbb{R}^{N\times D}$ is the matrix of activations in the l^{th} layer while $H^{(0)} = X$.

Then, a supervised node classification problem can be defined using a n-layer GCN:

$$Z = f(X, A) = \mathbf{softmax}(\hat{A} \ldots \mathbf{ReLU}(\hat{A}XW^{(0)}) \ldots W^{(n)}) \tag{3}$$

Here, $\hat{A}$ denotes $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$. The **softmax** classifier [36] is applied row-wise. In practice, we choose to use a two-layer structure (i.e., the hyperparameter $n = 2$) due to the efficiency consideration. Thus, $W^{(0)} \in \mathbb{R}^{C\times H}$ is an input-to-hidden weight matrix where the input has C channels (i.e., number of features in a node) and the hidden layer has H feature maps; $W^{(1)} \in \mathbb{R}^{H\times 1}$ is a hidden-to-output weight matrix. Besides, to prevent overfitting, we also apply a $\mathbf{dropout}(\cdot)$ function [60] to the output of each hidden layer and the output layer.

The weights $W^{(n)}$ can be trained via optimizing the following loss function, which evaluates the binary cross-entropy error over labeled examples:

$$\mathcal{L} = -\sum_{v_i \in \mathcal{V}} Y_{v_i} \ln Z_{v_i} \tag{4}$$

where Z_{v_i} represents the predicted probability of node v_i being security-related, and Y_{v_i} denotes its ground-truth label: 1 for being security-sensitive; 0 otherwise.

E DESCRIPTION GENERATION FROM FTGS

With the defined language FTL, we can translate an entire FTG to descriptive texts. Algorithm 2 illustrates our algorithm.

More concretely, our algorithm DESCGEN() takes a FTG and a semantic model M as inputs and generates the textual description $\mathbb{DESC}$. $\mathbb{DESC}$ is initialized to be an empty set and eventually consists of multiple sentences. Given a graph, we first compute a subgraph $\mathsf{FTG}_{\mathsf{ctrl}}$ containing only the "control transfer" edges, and then iterate over every node on each path p of $\mathsf{FTG}_{\mathsf{ctrl}}$ to produce a single-sentence description desc.

If a node is a conditional statement, we will find the definitions of the predicate variables (left-hand side and right-hand side), from the *"other data origin"* nodes ($\mathsf{DEF}_{\mathsf{node}}$) that this condition depends on, and leverage our semantic model to concretize the data-origin and data-type of the definitions so as to generate subject (left-hand side) and object (right-hand side) phrases. Then, with the concretized {subj,ojb} and the predefined verb *"be equal to"*, we can produce a condition modifier, which is further aggregated into the sentence.

Otherwise, if the node is either a *transfer call* or a *nearest common data origin* (NCDO), the subject is the smart contract *"function"* which performs this action. Then, we use the data dependencies – i.e., definitions $\mathsf{DEF}_{\mathsf{node}}$ and uses $\mathsf{USE}_{\mathsf{node}}$ of this node – along with the semantic model M to concretize the object. For instance, if a variable calculated in a node originates from a *constant* and is further used as a transferred *amount*, it is thus translated into *"the 'constant' 'amount' "*. Besides, we will also select the corresponding verb (*"transfer"* or *"calculate"*) and modifier (*"from ... to ..."* or *"using"*), based upon the node attribute, and then concretize the modifier using the data origins of this node.

In our implementation, we use SimpleNLG [52] to realize and aggregate sentences. SimpleNLG performs general optimizations to make sentences more concise. For example, it aggregates multiple sentences sharing the same subject. In addition, we also conduct custom optimizations to merge sentences that significantly overlap one other.

Table A1: API Models

API Prototype	Parameter Type	Source
Blockhash(unit blocknumber) returns (bytes32)	Blockhash(UINT) returns (BYTES)	Solidity
block.basefee	UINT	Solidity
block.chainid	UINT	Solidity
block.coinbase	ADDRESS	Solidity
block.difficulty	UINT	Solidity
block.gaslimit	UINT	Solidity
block.number	UINT	Solidity
block.timestamp	TIMESTAMP	Solidity
gasleft()	UINT	Solidity
msg.data	BYTES	Solidity
msg.sender	ADDRESS	Solidity
msg.value	AMOUNT	Solidity
msg.sig	BYTES	Solidity
tx.origin	ADDRESS	Solidity
tx.gasprice	UINT	Solidity
require(bool condition)	BOOL	Solidity
assert(bool condition)	BOOL	Solidity
revert()	NO TYPE	Solidity
addmod(uint x, uint y, uint k) returns (uint)	addmod(UINT, UINT, UINT) returns (UINT)	Solidity
mulmod(uint x, uint y, uint k) returns (uint)	mulmod(UINT, UINT, UINT) returns (UINT)	Solidity
keccak256(bytes memory) returns (bytes32)	keccak256(BYTES) returns (BYTES)	Solidity
ripemd160(bytes memory) returns (bytes20)	ripemd160(BYTES) returns (BYTES)	Solidity
sha256(bytes memory) returns (bytes32)	sha256(BYTES) returns (BYTES)	Solidity
address.balance	BALANCE	Solidity
address.codehash	BYTES	Solidity
address.send(uint256 amount) returns (bool)	ADDRESS.send(AMOUNT) returns (BOOL)	Solidity
address.transfer(uint256 amount)	ADDRESS.transfer(AMOUNT)	Solidity
address.delegatecall(bytes memory)	ADDRESS.delegatecall(BYTES)	Solidity
transferFrom(address _from, address _to, uint256 _value) returns (bool)	transferFrom(ADDRESS, ADDRESS, AMOUNT) returns (BOOL)	ERC-20
transfer(address _to, uint256 _value) returns (bool)	transfer(ADDRESS, AMOUNT) returns (BOOL)	ERC-20
approve(address _spender, uint256 _value) returns (uint256)	approve(ADDRESS, AMOUNT) returns (UINT)	ERC-20
allowance(address _owner, address _spender) returns (uint256)	allowance(ADDRESS, ADDRESS) returns (UINT)	ERC-20
balanceOf(address _owner) returns (uint256)	balanceOf(ADDRESS) returns (UINT)	ERC-20
totalSupply() returns (uint256)	totalSupply() returns (AMOUNT)	ERC-20
granularity() returns (uint256)	granularity() returns (UINT)	ERC-777
balanceOf(address owner) returns (uint256)	balanceOf(ADDRESS) returns (BALANCE)	ERC-777
send(address recipient, uint256 amount, bytes data)	send(ADDRESS, AMOUNT, BYTES)	ERC-777
burn(uint256 amount, bytes data)	burn(AMOUNT, BYTES)	ERC-777
isOperatorFor(address operator, address tokenHolder) returns (bool)	isOperatorFor(ADDRESS, ADDRESS) returns (BOOL)	ERC-777
authorizeOperator(address operator)	authorizeOperator(ADDRESS)	ERC-777
revokeOperator(address operator)	revokeOperator(ADDRESS)	ERC-777
defaultOperators() returns (address[] memory)	defaultOperators() returns (ADDRESS_ARRAY)	ERC-777
operatorSend(address sender, address recipient, uint256 amount, bytes data, bytes operatorData)	operatorSend(ADDRESS, ADDRESS, AMOUNT, BYTES, BYTES)	ERC-777
operatorBurn(address account, uint256 amount, bytes data, bytes operatorData)	operatorBurn(ADDRESS, ADDRESS, BYTES, BYTES)	ERC-777
allowance(address _owner, address _spender) returns (uint256)	allowance(ADDRESS, ADDRESS) returns (UINT)	ERC-777
approve(address _spender, uint256 _value) returns (uint256)	approve(ADDRESS, AMOUNT) returns (UINT)	ERC-777
transferFrom(address holder, address recipient, uint256 amount)	transferFrom(ADDRESS, ADDRESS, AMOUNT)	ERC-777
_mint(address account, uint256 amount, bytes userData, bytes operatorData, bool requireReceptionAck)	_mint(ADDRESS, ADDRESS, AMOUNT, BYTES, BYTES)	ERC-777
_send(address from, address to, uint256 amount, bytes userData, bytes operatorData, bool requireReceptionAck)	_send(ADDRESS, ADDRESS, ADDRESS, AMOUNT, BYTES, BYTES, BOOL)	ERC-777
_burn(address from, uint256 amount, bytes data, bytes operatorData)	_burn(ADDRESS, ADDRESS, AMOUNT, BYTES, BYTES)	ERC-777
_approve(address holder, address spender, uint256 value) returns (uint256)	_approve(ADDRESS, ADDRESS, AMOUNT) returns (uint256)	ERC-777
_callTokensToSend(address operator, address from, address to, uint256 amount, bytes userData, bytes operatorData)	_callTokensToSend(ADDRESS, ADDRESS, ADDRESS, AMOUNT, BYTES, BYTES)	ERC-777
_callTokensReceived(address operator, address from, address to, uint256 amount, bytes userData, bytes operatorData, bool requireReceptionAck)	_callTokensReceived(ADDRESS, ADDRESS, ADDRESS, AMOUNT, BYTES, BYTES, BOOL)	ERC-777