Automated Generation of Security-Centric Descriptions for Smart Contract Bytecode

Yu Pan
University of Utah
Salt Lake City, UT, USA
yupan97@cs.utah.edu

Zhichao Xu
University of Utah
Salt Lake City, UT, USA
brutusxu@cs.utah.edu

Levi Taiji Li
University of Utah
Salt Lake City, UT, USA
levili@cs.utah.edu

Yunhe Yang
University of Utah
Salt Lake City, UT, USA
yunhe.yang@utah.edu

Mu Zhang
University of Utah
Salt Lake City, UT, USA
muzhang@cs.utah.edu

ABSTRACT

Smart contract and DApp users are taking great risks, as they do not obtain necessary knowledge that can help them avoid using vulnerable and malicious contract code. In this paper, we develop a novel system *Tx2TXT* that can automatically create security-centric textual descriptions directly from smart contract bytecode. To capture the security aspect of financial applications, we formally define a *funds transfer graph* to model critical funds flows in smart contracts. To ensure the expressiveness and conciseness of the descriptions derived from these graphs, we employ a *GCN*-based model to identify security-related condition statements and selectively add them to our graph models. To convert low-level bytecode instructions to human-readable textual scripts, we leverage robust API signatures to recover bytecode semantics. We have evaluated *Tx2TXT* on 890 well-labeled vulnerable, malicious and safe contracts where developer-crafted descriptions are available. Our results have shown that *Tx2TXT* outperforms state-of-the-art solutions and can effectively help end users avoid risky contracts.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

decentralized apps; smart contracts; textual description; program analysis; natural language generation

ACM Reference Format:
Yu Pan, Zhichao Xu, Levi Taiji Li, Yunhe Yang, and Mu Zhang. 2023. Automated Generation of Security-Centric Descriptions for Smart Contract Bytecode. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598132

ISSTA 2023, 17-21 July, 2023, Seattle, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598132

1 INTRODUCTION

Smart contracts are autonomous computer programs running atop blockchains. They have the unique ability to enable trustworthy and decentralized transactions, and thus have become the enabling techniques for popular decentralized applications (DApps), such as major NFT marketplaces [6, 13] and emerging decentralized finance (DeFi) [4, 16]. The monthly transaction volumes of these applications are in billions of US dollars [7].

In the meantime, DApp end users are taking great risks. Smart contracts are known to have many security issues and logic errors [2, 25, 31, 33, 37, 38, 43, 46, 48, 49, 55, 59], which can lead to drastic financial losses. In contrast, app users have very little knowledge about the contract code they are running – app UI may provide high-level textual descriptions of contract behaviors (e.g., auction, token swap) but does not speak to concrete implementations of contract logic where security and safety risks actually reside. Without necessary information about security threats in underlying smart contracts, end users cannot make any informed decisions to rationally avoid using risky contracts.

Existing smart contract security analyzers [8, 9, 11, 12, 21] can automatically describe detected risks in natural language scripts based on predefined templates. However, they focus on individual low-level security problems such as reentrant functions [46] or integer overflow [49] but do not explain how these general problems in computer programs can affect end users' financial security in specific transaction contexts. In contrast, natural language processing-based techniques [41, 64] can learn a model from smart contract source code so as to summarize transaction logic in concrete contexts. Nevertheless, they heavily rely on symbol information and developers' comments which are neither trustworthy nor always available.

To address these limitations, we propose to automatically generate textual descriptions of smart contracts – directly from their bytecode – to inform end users of whether and how these computer programs put their funds in risk.

While little has been done to describe security risks in smart contract bytecode, the similar idea has been implemented in other domains such as Android [27, 65, 66] or IoT apps [57]. To model critical app behaviors, their descriptions are built around API names whose security implication can be easily understood by end users. For instance, the usage of `requestLocationUpdates()` implies that an app may track the user's location history; an API call to `lock.unlock` suggests the user may be at the risk of burglary.

Nevertheless, describing critical API calls in smart contracts, such as the funds transfer functions of Solidity or ERC-20, is insufficient to raise an alert to smart contract users because they are commonly used in all kinds of financial applications – benign, flawed or malicious. The way, in which a financial transaction is made, matters. For example, a normal call to the `transfer()` API can be suddenly exploited to mount double-spending attacks [38, 46] if it is made within a reentrant function; an unfair sales practice may disregard even legitimate user payments [35, 44]; a fraudulent "honeypot" contract [58] can stealthily send a user's funds to an attacker's account; self-destructive and suicidal contracts [40] lead to financial losses because users have deposited funds but can never withdraw them. Hence, our key observation is that *multiple smart contracts using the same funds transfer APIs may or may not cause a security problem due to the different ways these calls are made.* Therefore, describing how funds transfers are made is necessary for end users to understand the security risks in smart contracts.

To solve this problem, we develop a tool *Tx2TXT* that can automatically distill funds transfer-related core semantics from smart contract bytecode and describe them to end users in a security-aware and human-comprehensible fashion. In particular, we *(a)* first develop custom static program analyses to selectively extract contract information directly from Solidity bytecode, and use this knowledge to build a *funds transfer graph* (`FTG`). The extracted graphs can be further improved by adding condition information. However, not all conditions are security-related. We thus *(b)* train a machine learning model to automatically identify critical preconditions for funds transfer activities. Finally, we *(c)* convert enhanced `FTG`s to natural language scripts. To this end, we recover high-level semantics from low-level bytecode so as to generate human-readable texts. Our produced descriptions are eventually used to complement existing security reports.

To the best of our knowledge, we are the first to bridge the gap between low-level implementations of smart contracts and human understanding of financial application logic.

We have implemented a prototype system in 1,500 lines of Python code. We have applied *Tx2TXT* to 890 well-labeled vulnerable, malicious and safe real-world contracts where developer-crafted descriptions are available, to evaluate its effectiveness. Our experimental results have shown that *Tx2TXT* can faithfully express essential smart contract behaviors and effectively cover critical security-related code content. Our user study has indicated that *Tx2TXT* outperforms the state-of-the-art solutions, and can successfully help average users avoid risky contracts.

In summary, this paper makes the following contributions:

- We propose a novel technique to protect the increasing population that uses DApps. To this end, we develop a tool to automatically generate security-centric descriptions for smart contract bytecode.
- We define a new graph model to capture the security semantics of financial applications using funds transfer activities.
- We address unique challenges in analyzing smart contract bytecode, so as to bridge the semantic gap between low-level representation and human readable descriptions.
- We have developed a prototype *Tx2TXT*. Our result shows that *Tx2TXT* outperforms existing descriptions from developers and security analyzers by a large margin.

```
1  contract DutchAuction {
2    uint listingTime; uint expirationTime;
3    uint deductedPrice; uint basePrice;
4    uint feeRate; address feeRecipient;
5    uint stolenAmount; address hacker;
6
7    function executeFundsTransfer(address token, address buyer,
        address seller) internal returns (uint) {
8      //Calculate the selling price based on the elapsed time
9      uint diff = SafeMath.div(SafeMath.mul(deductedPrice, SafeMath
        .sub(now, listingTime)), SafeMath.sub(expirationTime,
        listingTime));
10     uint price = SafeMath.sub(basePrice, diff);
11
12     //Transfer funds to the specified account
13     if (price > 0 && token != address(0))
14       ERC20(token).transferFrom(buyer, seller, price);
15
16     //Pay transaction fee proportional to transferred amount
17     uint fee = SafeMath.mul(price, feeRate);
18     ERC20(token).transferFrom(buyer, feeRecipient, fee);
19
20     //Malicious hidden transfer
21     ERC20(token).transferFrom(buyer, hacker, stolenAmount);
22   }
23 }
```

Figure 1: Dutch Auction with Hidden Funds Transfers

To facilitate further research, we are committed to make the source code and dataset publicly available.

2 PROBLEM & APPROACH

2.1 Motivating Example

We use a malicious auction contract as an example to motivate our work. This contract implements a Dutch auction [6] and contains a hidden transfer problem that has been studied by prior work such as HONEYBADGER [58] and TokenScope [26].

Figure 1 illustrates the source code of this contract `DutchAuction`, written in the Solidity. Specifically, this contract first defines multiple global variables that are used to set up an auction. These include *(1)* the start and end time of an auction, `listingTime` and `expirationTime`, *(2)* the base price of the merchandise `basePrice` and the gradually deducted amount `deductedPrice`, and *(3)* transaction fee rate `feeRate` as well as the fee recipient `feeRecipient`. In addition, this malicious contract defines the amount to be stolen for each transaction, `stolenAmount`, and the address of the `hacker` to receive the stolen funds.

Once a user makes an offer, the `executeFundsTranfer()` function (ln.7) is invoked to calculate the bid price and then transfer funds based upon the input addresses of the `buyer` and `seller`. Unlike the popular English auction where participants attempt to become the highest bidder, in a Dutch auction, the auctioneer starts with a high selling price and lowers it over time until some participant accepts the price. Thus, to obtain this final selling `price`, this function calculates a `diff` according to the elapsed time `SafeMath.sub(now, listingTime)` (ln.9), and subtracts the `diff` from the `basePrice` (ln.10). If the derived `price` is valid (i.e., greater than zero), the contract will transfer this amount of tokens from the `buyer` to the `seller` (ln.14). Furthermore, the contract will also pay the transaction `fee` based on the selling `price` to the `feeRecipient` from the `buyer`'s account (ln.18). Aside from these legitimate actions, in the end, the contract will stealthily send a specific amount of funds to the `hacker`'s account (ln.21).

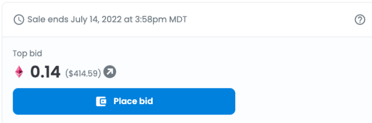


Figure 2: Unclear UI Text for Dutch Auction in OpenSea

> *The function calculates an* ***amount*** *using a timestamp and transfers this* ***amount*** *of tokens* ***from an input address to another input address****, and then calculates a* ***second amount*** *using the first amount and transfers this* ***amount*** *of tokens to* ***a third-party address****, and finally transfers a* ***third amount*** *to* ***another third-party address****.*

Figure 3: Expected Textual Description for the Example

Unfortunately, such nuances in smart contract implementations are not necessarily reflected on DApp front-end interfaces. For instance, the web UI of *OpenSea* (Figure 2), one of the most popular NFT market app [6], simply indicates that the app allows end users to “Place bid”, despite that it internally implements a non-trivial Dutch auction logic and calculates/transfers various interests and fees based upon very sophisticated business models.

Besides, being deployed to the blockchain, smart contract source code is compiled to obscure bytecode. Because all the symbols have been stripped from the bytecode executable, it becomes very difficult (if not impossible) for human readers to recognize the original logic of the program.

Admittedly, using automated program analysis, we can still identify robust API signatures such as `CALL`, `REVERT` or `TIMESTAMP` at the bytecode level, which are used to manage crucial funds transfers. However, the existence of such API calls is not a key differentiator between normal transactions and dangerous behaviors. In the motivating example, the same *ERC-20* API `transferFrom()` is used for both the benign funds/fee transfer (ln.14 and 18) and the malicious theft of user funds (ln.21). The difference lies in how a funds transfer is made. Particularly, in the first normal transaction (ln.14), the funds are transferred to a user specified input `buyer`, and the transferred amount `price` is calculated from a time factor `now` due to the nature of Dutch auctions. In the second normal transfer (ln.18), the transferred `fee` is calculated based upon the previously transferred amount `price`. In contrast, in the malicious transaction (ln.21), both the amount to be transferred `stolenAmount` and the funds recipient `hacker` are irrelevant to either the user request or the auction logic.

2.2 Problem Statement

Our Goal. To help end users understand the security risks in smart contract bytecode, we propose to automatically generate textual contract descriptions that capture important security semantics in specific funds-transfer contexts. For instance, in the motivational example, we hope to create a textual description shown in Figure 3. Such a description must capture the crucial (normal and abnormal) behaviors of this contract: *(1)* there exist three different funds transfers. Their difference lies in how transferred amounts and the funds recipients are obtained, indicated by the underlined texts. *(2)* The contract implements a Dutch auction – the selling price (i.e., the first **amount**) is derived using the `now` *timestamp*, and sent to an *input address*, the buyer account specified by the input. *(3)* An additional fee (i.e., the **second amount**) is required for this transaction and is calculated based upon the selling price (i.e., *first amount*). *(4)* The irrelevant parameters (**amount** and **address**) make the third funds transfer look suspicious.

Note that we do not intend to use our descriptions to replace existing textual reports generated by security analyzers. Detecting security risks is orthogonal to the goal of this work. Instead, our descriptions can complement the abstract reports via providing concrete funds-transfer contexts which are necessary for human readers to understand reported problems.

Design Requirements. To design a system that achieves our goal, several requirements must be met:

(1) **Security-centric.** We expect textual descriptions to help end users understand security risks in smart contracts. Thus, they must cover security-related contract behaviors.
(2) **Bytecode-oriented.** We must build descriptive scripts solely from smart contract bytecode. We must not use any additional information such as domain knowledge or heuristics.
(3) **Human-readable.** Readable textual descriptions must be succinct. Tedious texts can hinder effective communication processes.
(4) **Risk Avoidance.** Our descriptions must assist humans in avoiding security risks in smart contracts. They must provide specific funds-transfer information that can enable humans to understand concrete contract logic and hidden financial security problems.

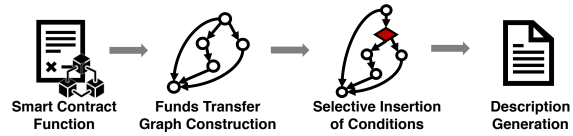


Figure 4: Architecture Overview of *Tx2TXT*

2.3 Approach Overview

To fulfill these requirements, we propose a novel technique *Tx2TXT* which can automatically extract security-related financial activities from smart contract bytecode and then translate them into human readable textual scripts. *Tx2TXT* consists of three major steps as shown in Figure 4.

(1) **Funds Transfer Graph Construction.** To model smart contract code in a security-aware fashion, we propose a novel graph representation *funds transfer graph* (`FTG`). To construct a `FTG` for a given smart contract function, we perform static control-flow and dataflow analyses to extract its intrinsic dependency information that indicates how user funds are transferred.
(2) **Selective Insertion of Security-Related Conditions.** Preconditions play an important role in understanding security risks of funds transfers. However, a large amount of conditions can greatly hinder the readability of `FTG`s and generated descriptions. Besides, not all the conditions are security relevant. To introduce additional security knowledge to our graph models while keeping descriptions concise, we train a machine learning model to automatically identify security-sensitive conditions, and then insert only these conditions into `FTG`s.
(3) **Description Generation.** To translate bytecode-level information into human-readable texts, we leverage robust API signatures to recover high-level semantics from low-level bytecode implementations. Based upon the unique semantic-level knowledge we can

obtain, we define a custom description language and develop a specific description generation algorithm that can convert `FTG`s to natural language scripts.

3 FUNDS TRANSFER GRAPH

3.1 Key Factors

To model security semantics in smart contracts, we argue that several key factors with respect to funds transfers must be taken into consideration.

- **Transfer API.** Funds transfer APIs such as `transfer()` of *Solidity*, `transferFrom()` of *ERC-20*, are required to enable transactions in financial applications. Because attackers in this domain aim for financial gain, they must exploit these functions to steal funds [26], double spend [28] or commit fraud [58].
- **Dataflow.** Knowing the presence of funds transfers is necessary but not sufficient. It is also critical to understand what has been transferred and where it is sent to. In the motivating example, we show that the source of funds and the funds recipient may indicate the legitimacy of a funds transfer – a "greedy" or "prodigal" [40] contract can withhold any arbitrary amount of funds and send them to attackers' accounts.
- **Relations Among Transfers.** As shown in our motivating example, financial services such as NFT markets [6] or token exchange [16] usually charge fees for each transaction and therefore commonly use multiple consecutive transfer APIs to send several corresponding amounts (i.e., transferred funds and fees). This can be reflected in the execution order and the shared data sources of multiple API calls. In contrast, an illegal transfer may not be relevant to any other legitimate funds transfers.
- **Specific Values.** First, constants play an important role in security analysis. Certain constant values (e.g., malicious account addresses) or even the presence of constant parameters such as funds recipients or transferred amounts can be an indicator of security risks. Second, a large portion of smart contracts implements a game of chance and therefore must depend on random numbers. It may raise security and fairness concerns if their key parameters do not rely solely on random number generators such as `keccak256()` [50].

3.2 Formal Definition

With the understanding of these key factors, we formally define the funds transfer activities in each contract function as a *Funds Transfer Graph* (`FTG`). A `FTG` depicts what and how cryptocurrency funds are being transferred.

Definition 1. A *Funds Transfer Graph* is a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \alpha, \beta)$ over a set of instructions Σ and relations $\mathcal{R}$, where:

• The set of vertices $\mathcal{V}$ corresponds to the instructions in Σ; these instructions include funds transfer API calls and those that these API calls depend on.

• The set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ corresponds to the control transfers or data dependencies among instructions.

• The labeling function $\alpha : \mathcal{V} \to \Sigma$ associates nodes with the labels of corresponding instructions. Each label consists of two elements: an instruction in a SSA-formed intermediate representation and an attribute. An attribute can be *"transfer call"*, *"nearest common data origin of transferred amounts"* or *"other data origin"*.

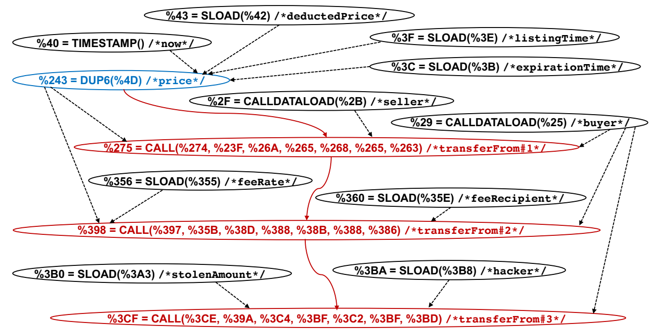


Figure 5: `FTG` of the motivating example. Red nodes are *"transfer calls"*; blue nodes are *"nearest common data origins of transferred amounts"*; black ones are the *"other data origins"*. Red solid curved lines represent control transfers; black dotted lines indicate data dependencies.

• The labeling function $\beta : \mathcal{V} \to \mathcal{R}$ associates edges with the labels of *"control transfer"* or *"data dependency"*.

3.3 `FTG` of Motivating Example

Figure 5 illustrates the `FTG` of our motivating example. Here, each node contains a bytecode instruction presented in the Octopus [1] IR, a SSA-formed representation of the original Solidity bytecode. For instance, `%43 = SLOAD(%42)` denotes that a variable `%43` is loaded from the *storage* space at the address `%42`. A `CALL` instruction `CALL(%gas,%addr,%wei,%in,%insize,%out,%outsize)` can invoke a function defined in a contract at address `%addr` with the given `%gas` and `%wei`. The function signature and parameters are passed through an input state `%in`, which will eventually be updated to become an output state `%out`. For readability purposes, we add a comment to each node to correlate every bytecode instruction with its source-level symbol. For example, `%40 = TIMESTAMP()` is associated with reading the `now` property in the source code, and the three `CALL` instructions correspond to the `transferFrom()` calls.

The red nodes are labeled with the *"transfer call"* attribute. The blue node `%243 = DUP6(%4D)`, which defines the `price` variable, is the *"nearest common data origin"* of two transfer calls and indicates the data dependency between the "amount" parameters of the two function calls. The black nodes represent the *"other data origins"* of the red and blue nodes. For instance, `transferFrom#3` depends on the data items of three black nodes, a buyer address, a hacker address and a stolen amount. The first originates from a function argument (i.e., `CALLDATALOAD`) and the last two are obtained from global variables loaded from *storage* via `SLOAD`.

The red curved lines represent control transfers. They show the three consecutive calls to the `transferFrom()` API and that the calculation of the bid `price` happens prior to these calls. The black dotted lines indicate data dependencies. One node can depend on multiple data sources. For instance, the `price` is calculated from the data inputs from four other nodes: the `now` property, `listingTime`, `expirationTime` and `deductedPrice`. The same data origin can affect multiple nodes. For example, the `buyer` address is used in all three transfer calls as the source of funds. The relation between two nodes can involve both control transfer and data dependency – the calculation of selling price occurs before the sequence of transfers and also has impact on the transferred amount of `transferFrom#1`.

Essentially, this graph captures all the important information that can facilitate user understanding of the contract logic: *(1)* there exist three funds transfer activities; *(2)* while all of them transfer funds from the buyer account, they differ in the transferred amounts and funds recipients; *(3)* in the first transaction, the transferred amount is calculated based upon a timestamp and other global variables while the recipient is specified by the user input; *(4)* in the second one, the funds recipient is designated by the contract but the transferred amount depends on the previously calculated price; *(5)* in contrast, neither the amount nor the recipient is relevant to any user inputs in the third transfer.

3.4 Graph Construction

We develop our custom static analysis to build the `FTG` for each contract function. Our analysis tool is built on top of Octopus [1] and can perform context-sensitive, flow-sensitive, inter-procedural dataflow analysis on Solidity bytecode.

Algorithm. Algorithm 1 describes how we construct a `FTG`. Specifically, our algorithm `BuildFTG()` takes a smart contract function `func` as an input and outputs a `FTG` graph. At the initialization stage, it first creates an empty edge set `FTG` (ln.2), and computes the control flow graph `CFG` of the given `func` (ln.3), and then collects all the `transfer` function calls in this function and stores them into a set $\mathbb{TC}$ (ln.4). For each pair of `transfer` calls $(\mathtt{tc}, \mathtt{tc}')$ in $\mathbb{TC}$, we invoke `FindNearestCommonDataOrigin()` to identify the shared definitions $\mathbb{NC}$, of their parameters of transferred amounts, that are closest to the callsites (ln.5-7).

More concretely, given a pair of calls $(\mathtt{tc}, \mathtt{tc}')$, we obtain the pair of their "amount" parameters $(\mathtt{amt_{tc}}, \mathtt{amt_{tc'}})$, and compute their use-define chains $\mathbb{DEF}_{\mathtt{tc}}$ and $\mathbb{DEF}_{\mathtt{tc'}}$, respectively (ln.21,22). Then, we will return the last definition of their intersection (ln.23), which will be the *nearest common data origin* of the two calls.

We further remove everything except for the instructions in $\mathbb{NC}$ and $\mathbb{TC}$ from the `CFG`. The remaining nodes, connected by control-transfer edges, will be added to the `FTG` (ln.8). Then, for each variable "use" used in either $\mathbb{TC}$ or $\mathbb{NC}$, we perform a use-define chain analysis to obtain all the definitions (*other data origins*) (ln.10). If a definition is an "external" one, meaning it is defined using a global variable, a constant, an API return value or a function parameter, we insert this definition as a new node to the `FTG` and add an edge from this node to the one using this definition (ln.11-15). Finally, the algorithm returns the `FTG` (ln.17).

Special Challenge. To implement our algorithm for Solidity bytecode, we need to address the unique calling convention and memory modeling used in the argument passing. As indicated in Figure 5, a function call to an ERC API (e.g, `ERC20.transferFrom()`) in Solidity source code is eventually compiled to a `CALL(%gas,%addr,%wei,%in,%insize,%out,%outsize)` instruction. However, instead of explicitly specifying a call target and passing each parameter to this function, in the `CALL` instruction, the function signature and all the arguments are implicitly passed through a `memory` region, at an address specified by the `%in` parameter. As a result, to identify the exact calls to make funds transfers and distinguish the data origins of individual parameters, we must understand this memory model.

Algorithm 1 Graph Construction

```
1:  procedure BuildFTG(func)
2:      FTG ← ∅
3:      CFG ← BuildCFG(func)
4:      𝕋ℂ ← GetTransferCalls(func)
5:      for ∀(tc, tc′) ∈ 𝕋ℂ do
6:          ℕℂ ← FindNearestCommonDataOrigin(tc, tc′)
7:      end for
8:      FTG ← CFG ∩ (𝕋ℂ ∪ ℕℂ)
9:      for ∀use ∈ (𝕋ℂ ∪ ℕℂ) do
10:         𝔻𝔼𝔽 ← DoUseDefChainAnalysis(use)
11:         for ∀def ∈ 𝔻𝔼𝔽 do
12:             if IsExternal(def) then
13:                 FTG ← FTG ∪ < def, use >
14:             end if
15:         end for
16:     end for
17:     return FTG
18: end procedure
19:
20: procedure FindNearestCommonDataOrigin(tc, tc′)
21:     𝔻𝔼𝔽_tc ← DoUseDefChainAnalysis(amt_tc)
22:     𝔻𝔼𝔽_tc′ ← DoUseDefChainAnalysis(amt_tc′)
23:     return GetLastDef(𝔻𝔼𝔽_tc ∩ 𝔻𝔼𝔽_tc′)
24: end procedure
```

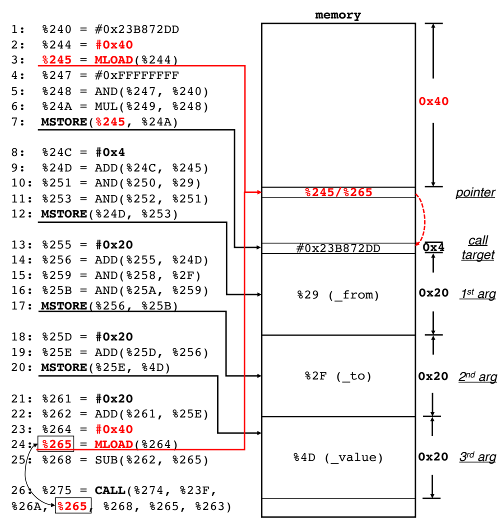


Figure 6: Argument Passing in Bytecode

Figure 6 demonstrates how parameters are passed for a call to `transferFrom(address _from, address _to, uint256 _value)`. In general, the caller and the callee access the same memory space containing the parameters based upon an implicitly agreed way for memory referencing. By convention, Solidity manages memory using a "free memory pointer" [51] at offset `0x40` in `memory`. Here, both the caller and the callee load this pointer (`%245/%265`) from this position (ln.3 and ln.24), and use it to locate the beginning of the memory allocated for parameters (ln.7 and ln.26). Specifically, the caller first stores a four-byte value (`#0x23B872DD`) at the top of this space (ln.7). This value is the "function selector" [19] that specifies which function to call, while `#0x23B872DD` is the signature of the `transferFrom()` function defined in *ERC-20* [18]. Then, the caller further pushes three arguments of this function, `_from`, `_to` and `_value`, each taking `0x20` bytes, consecutively into the following spaces (ln.12,17,20). Finally, the `CALL` instruction passes the address (`%in`) and the size (`%insize`) of this memory cell to the callee (ln.26), and the callee will parse this memory region to identify the call target as well as its corresponding parameters.

To address this implicit argument passing, we first use backward dataflow analysis to discover the source (`MLOAD`) of the `%in` parameter in a `CALL` instruction. This source, pointed to by a "free memory pointer", is the address of the memory holding function arguments. Then, we identify the previous `MLOAD` instruction that can obtain the same pointer value. To this end, we perform a simple points-to analysis to check whether the specific memory cell at offset `#0x40` can be updated between two `MLOAD`s. Starting from the previous `MLOAD`, we conduct forward dataflow analysis to discover a series of memory writes (`MSTORE`) that store arguments sequentially into the memory space. For each `MSTORE`, we compute its data sources to find the content of a specific argument being passed.

4 SECURITY-RELATED CONDITIONS

While a basic `FTG` depicted in Figure 5 can already capture key funds transfer activities, the graph model can be more expressive if it also explains the circumstances under which funds transfers are made. Certain condition checks have strong security implications – if a necessary safety check (e.g., balance or expiration check) is absent or an uncommon check (e.g., backdoor, logic bomb or unsatisfiable condition [58]) is present, a funds transfer guarded by such a condition may look more suspicious.

A naïve approach to considering the impact of conditions would be adding all relevant conditional statements to `FTG`s based upon the control flow graph. Nevertheless, this may lead to a significant growth of graph (and eventually description) sizes. In fact, not all conditions are security-sensitive. For example, the condition check in the motivating example simply confirms that the selling price, an unsigned integer, is greater than zero. While doing so can avoid unnecessary transactions, the contract will not cause any security problems even without such a check. In order to build security-centric descriptions, we hope to selectively insert only security-related condition checks into our graphs.

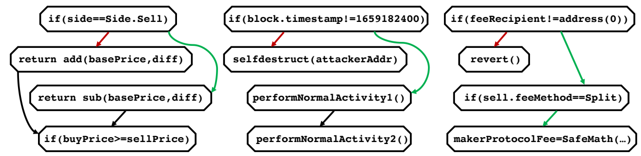
Note that, to be safe, when identifying security-related conditions, we take a conservative approach. Our technique is designed to be "recall-oriented" – we expect to see no false negatives but may cause over-approximation. However, our trade-off would at most result in larger (but still correct) graphs.

4.1 Key Insights

Intuitively, the distinction between a security-sensitive condition and a less interesting one lies in *the form of predicates*, *the topology of conditional branches*, and *the consequence of condition checks*.

First, a harmful predicate, present in for example logic bombs or backdoors, often checks a runtime state – such as a user input, current time or system environment – against a narrow and constant value range [32], and thus is unlikely to compare a random local variable with an arbitrary value.

Second, because a security-related condition check often causes a program to enter two drastically different states (e.g., normal activities vs. self-destructing code in Figure 7(b), or authentication success vs. authentication failure), its two branches can be easily unbalanced. In contrast, if a conditional statement is related to regular program logic, its two branches, though exercising different code,



(a) Balanced Branches (b) Unbalanced Branches due to Logic Bomb (c) Unbalanced Branches due to Input Validation

Figure 7: Balanced vs. Unbalanced Branches

are likely to share the same intention (e.g., to calculate a selling price yet using different algorithms in Figure 7(a)) and therefore may result in similar lengths.

Third, normal condition checks may also contain unbalanced branches, exemplified by Figure 7(c), where the condition check is performed to implement an input validation. Nevertheless, a failed input check would at most revert a transaction, as opposed to harmful funds transfers caused by successfully triggered logic bombs.

As a result, to identify conditions that bear interesting security semantics, the context in which these conditions are checked matters. Nevertheless, such contextual differences are hidden in implementation nuances that cannot be easily expressed using simple code patterns. Besides, the aforementioned factors may be entangled and can be of different importance when assessing the security-relatedness of a condition. Hence, to automatically and accurately discover crucial condition checks, we train a deep learning model to quantitatively capture these subtle differences.

4.2 Basic Algorithm: GCN

To solve this node classification problem, we train a Graph Convolutional Network (GCN) [36] that can automatically determine if a given condition node is of security interest. Then, we can insert only these nodes and relevant control-transfer edges to `FTG`s.

Note that, any machine learning models that enable node classifications may potentially serve our needs. The usage of a GCN model is a demonstration of our node selection technique. Finding the best machine learning algorithms, that can most precisely capture security-relatedness of conditional statements, is orthogonal to the major goal of this work – which is to create human readable, security-centric contract descriptions.

The input of our GCN model is an annotated control flow graph (ACFG), which is formally defined in prior work [30, 62]. In general, an ACFG is an enhanced control flow graph where each node (basic block) is annotated with a vector of semantic-level features such as number of calls or instructions. The details of GCN can be found in online appendix D.

4.3 Semantics & Context-Aware Node Embedding

We then engineer the node features to encode smart contract-specific semantics and context into our GCN structure. Specifically, we first create a feature vector for each factor, and then concatenate all the vectors to generate a node embedding.

Semantics. We consider the semantics of each node to be indicated by the existence of API calls, constants, global variables, and conditions. Thus, we encode them into a feature vector where each dimension represents the presence of a corresponding feature.

Dependency Context. While a vanilla GCN can efficiently represent neighboring contexts of individual nodes, it does not capture their causal dependencies as it does not consider edge directions. Hence, we alternatively incorporate the direction (or dependency) information into node embeddings. To this end, we utilize the TransE [23] method which converts relationships of multi-relational data to low-dimensional vector spaces.

The core idea of TransE is to consider relationships as translations in the embedding space. Thus, for every edge, denoted as a triplet <*head node*, *edge label*, *tail node*> or to use our notions $< v_h, e, v_t >$, the embedding of the tail v_t should be close to that of the head v_h plus the vector of the edge e. The intrinsic dependency relations in a graph are hence captured by the optimal embeddings selection, which can be trained by first randomly initializing node and edge embeddings and then optimizing the following objective:

$$\sum_{(v_h, e, v_t) \in \mathcal{G}} ||emb_{v_h} + emb_e - emb_{v_t}|| \tag{1}$$

where emb_{v_h}, emb_e and emb_{v_t} denote the d-dimensional vectors of the head v_h, edge e and tail v_t, respectively. d is a configurable parameter; in practice we set it to be 100.

Topological Context. Furthermore, motivated by Figure 7 (a) and (b), we also hope to encode the knowledge of "the balance between two branches" into our model, so as to differentiate normal condition checks from special ones. However, such a high-level topological feature cannot be easily captured by a vanilla GCN because the basic model only looks at neighboring nodes within a constant distance of each central node. Although configurable, this distance in practice is often small for the sake of runtime performance. Consequently, we additionally include this information into node features. Particularly, we compute the difference between the lengths of two branches starting from every conditional statement, and use logarithmic encoding to generate its embedding to avoid sparse feature vectors. We consider the end of a branch to be the next conditional statement or the end of a function. For non-condition nodes, their feature vectors will be all zeroes.

4.4 Training Data

To train our model, we need to collect smart contract samples which contain well-labeled malicious or suspicious condition statements. However, to the best of our knowledge, there exists no such dataset to date. To address this problem, we instead collect relevant datasets in other domains. These benchmarking projects, though using logic bombs as demonstration, in fact systematically summarize malicious and suspicious "narrow" conditions such as time, system resources, system properties, random numbers or specific user inputs, and thus can broadly capture the semantics and contexts of security-sensitive condition checks in different scenarios including but not limited to logic bombs, backdoors, unreachable code, etc. We collect relevant datasets in other domains, and thus learn the model using a mixture of 562 C [61] and Android [14, 15, 22] logic bomb samples from benchmarking projects and top smart contract code retrieved from Etherscan [3] that are confirmed to be safe by Slither [2].

4.5 Insertion of Condition Nodes

Once we have identified security-related condition statements in a function, we add them to the corresponding `FTG` following the original control flow graph. To this end, we introduce a new node attribute *"condition"* and connect these condition nodes to existing graph nodes using *"control transfer"* edges. Additionally, we compute data sources of the variables used in conditions, and insert them as *"other data origins"* to the graph via adding a *"data dependency"* edge from each origin to its associated condition.

Table 1: Partial Semantic Entities

Semantics	Category	Source	Example
address	Explicit	API param or return	`<address>.send(),CALLER`
amount	Implicit	API param or return	`transferFrom(...,value),CALLVALUE`
balance	Implicit	API return value	`ERC20.balanceOf(),BALANCE`
timestamp	Implicit	API return value	`TIMESTAMP`
input	Origin	API return value	`CALLDATA`
random	Origin	API return value	`keccak256(),sha256(),ripemd160()`
global var	Origin	`storage/memory`	`SLOAD,MLOAD`
constant	Origin	constant value	`0x001d3f1ef827552ae111...`

5 DESCRIPTION GENERATION

In general, we follow prior work's [66] approach to convert our `FTG`s to natural language scripts. However, unlike the prior work which simply relies on semantic-rich API names to produce natural language elements – e.g., translating `sendTextMessage()` to a verb "send" and an object "text messages" – we must address the unique challenge of the semantic gap between low-level Solidity bytecode, such as `DUP1` or `SLOAD`, and human-understandable textual tokens such as "an input address" or "a constant transferred amount".

5.1 Semantic Modeling

To recover the semantics of Solidity bytecode instructions, we resort to three categories of robust information. Table 1 lists, in part, key semantic entities we can obtain from them.

(1) Explicit Data Type. Because smart contracts are particularly used to implement blockchain-based financial applications, a special data type `address` is introduced in Solidity to handle the unique feature of underlying platforms and transaction processes. An `address` variable essentially is a 20-byte Keccak-256 number but can represent an individual account used to maintain crypto assets. We can reliably obtain this type information from either API parameters or return types. For instance, the Solidity API `send()` uses an `address` as the target account. The return value of several API functions such as `msg.sender`(bytecode `CALLER`), `tx.origin(ORIGIN)`, `block.coinbase(COINBASE)` is an Ethereum `address`.

(2) Implicit Data Type. In addition to explicit native types, there also exists implicit yet finer-grained type information. Since variables of the same primitive type can be used for very different purposes, they in fact can bear distinctive semantic meanings. For example, an unsigned integer (`uint`) can be used to represent the amount of crytocurrency or tokens being transferred, or to indicate the balance of an account, or even to denote the current timestamp. The differentiation of these seemingly similar integers is of critical importance to precisely interpreting contract logic. To this end, we build a semantic model for every well-known Solidity and ERC API (a list in Table A1 in the online appendix [20]), and use it to infer the semantics of relevant variables. As exemplified in Table 1, the return value of a call to `CALLVALUE` or the third parameter of `ERC20.transferFrom()` is a transferred *amount*; the return value of either Solidity API `BALANCE` or ERC-20 function `balanceOf()`

```
⟨description⟩   ::= ⟨sentence⟩*
⟨sentence⟩      ::= ⟨sentence⟩ ‘, and then’ ⟨sentence⟩
                  | ⟨statement⟩ ⟨modifier⟩
⟨statement⟩     ::= ⟨subject⟩ ⟨verb⟩ ⟨object⟩
⟨subject⟩       ::= ⟨noun phrase⟩
⟨verb⟩          ::= ‘transfer’ | ‘calculate’ | ‘be equal to’
⟨object⟩        ::= ⟨noun phrase⟩
⟨modifier⟩      ::= ⟨modifier⟩ ⟨conj⟩ ⟨modifier⟩
                  | ‘if’ [‘not’] ⟨sentence⟩
                  | ⟨with⟩ ⟨noun phrase⟩
                  | ⟨empty⟩
⟨noun phrase⟩   ::= ⟨data-origin⟩ ⟨data-type⟩
                  | ⟨data-origin⟩
                  | ⟨data-type⟩
⟨conj⟩          ::= ‘and’ | ‘or’ | ⟨empty⟩
⟨with⟩          ::= ‘from’ | ‘to’ | ‘using’
⟨data-origin⟩   ::= ‘input’ | ‘constant’ | ‘global’ | ‘random’ |
                    ⟨ordinal⟩
⟨data-type⟩     ::= ‘address’ | ‘amount’ | ‘balance’ |
                    ‘timestamp’ | ‘value’
⟨ordinal⟩       ::= ‘1st’ | ‘2nd’ | ‘3rd’ | ...
⟨empty⟩         ::= ‘ ’
```

Figure 8: An Abbreviated Syntax of FTL

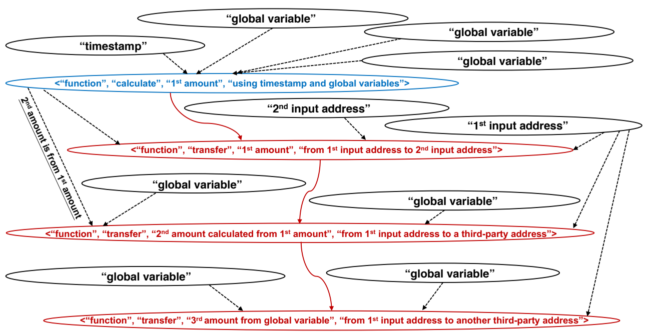


Figure 9: Translating `FTG` of Motivating Example

is an account *balance*; the `TIMESTAMP` call returns a *timestamp* variable.

(3) Data Origin. Aside from the type of a variable, we can also retrieve its origin. This serves as additional information that may help human users further distinguish variables of the same types. In the motivating example, the same `amount`-typed variables are used in multiple `transferFrom()` calls. However, depending on where these amounts originate, a user may infer whether a call is ill-intentioned. The knowledge about data origins can also be collected from modeling API return values (Table 1). For example, `CALLDATA` returns function *inputs* provided by contract users; hash functions `keccak256()`, `sha256()`, `ripemd160()`, etc. are the origins of *random* numbers. Additionally, *global variables* can be fetched from `storage` or `memory` via special instructions `SLOAD` or `MLOAD`. *Constants* are acquired from fixed numbers.

5.2 Funds Transfer Language

With the enhanced semantics, we formally define a funds transfer language (FTL) that can specifically capture transfer-related actions and key fund flows in our `FTG`s. Figure 8 depicts the abbreviated syntax of our language in Extended Backus-Naur form (EBNF).

In particular, a `description` of funds transfers consists of multiple `sentences`, each of which can be either recursively defined or directly formed as a `statement` plus a `modifier`. A `statement` indicates the activity that a `subject` performs (`verb`) on an `object`. A `modifier` specifies how an activity is performed – on what condition (`'if'`), depending on what data (`'using'`), or through what dataflow (`'from'` and/or `'to'`). Multiple `modifiers` can be concatenated directly or using a logical conjunction.

Both a `subject` and an `object` can be a `noun phrase`, which in our context, is composed of a `data-origin` and a `data-type`. The `data-origin` and `data-type` are derived from our semantic models. Specifically, for data origins, in addition to those that can be directly obtained from our models (e.g., `input`, `constant`), we further differentiate variables from different origins using `ordinal` numbers such as *“the first amount”* or *“the second address”*. For data types, we also introduce a basic `value` type for those whose types cannot be resolved using our semantic model. Besides using `data-origin` and `data-type` individually, we can also use the former as an attribute of the latter to form a compound phrase such as a *“constant address”* or an *“input amount”*.

We support three types of actions indicated by the `verb`. First, we describe the funds transfer activities using `'transfer'`. Second, we use `'calculate'` to illustrate the calculation of intermediate amount values. Last, we also introduce the `'be equal to'` to depict the comparison in conditional statements (i.e., the `'if'` clause). The details of description generation are in online appendix E.

5.3 Motivating Example

Figure 9 demonstrates how we convert `FTG` nodes to corresponding natural language elements for the motivating example.

Particularly, in this example, we identify one path (red) that represents the control flow of its transfer activities and thus needs to be described. For each node on the path, we convert it to natural language elements based on its attribute and data dependencies.

For instance, the first node (i.e., the blue one) yields the transferred amount in the first transfer call. Therefore, we use a `verb` *“calculate”* to describe this action and use the `data-type` *“amount”* as the `object` for this verb. The `modifier` represents how this amount is being calculated and is derived from the `data-origin` of this node – the timestamp and global variables.

All the other three nodes on the path are transfer calls, and therefore are illustrated using the `verb` *“transfer”* with the `object` *“amount”*. Depending on their individual data origins, these three `objects` are further enhanced in different ways. Specifically, the amount being transferred in the second call is calculated from the one used in the first call. To indicate this relation, we use ordinal numbers to differentiate these two *“amounts”* while adding a `modifier` *“calculated from 1st amount”* to define the 2nd amount. In contrast, the *“amount”* in the third call originates from a *“global variable”*, which shows it is totally irrelevant.

The sentences generated for the three transfer calls are also modified by their funds flows, described using the *“from” <sender address> “to” <recipient address>* `modifiers`. These `modifiers` are concretized based upon the origins and types of the addresses. While the first call obtains both sender and recipient addresses from *"inputs"*, the other two send funds to *"a third-party address"* (i.e., unknown global variable).

Finally, we can create this descriptive script for the example: *The function calculates 1st amount using timestamp and global variables, and then transfers 1st amount from 1st input address to 2nd input address, and then transfers 2nd amount calculated from 1st amount from 1st input address to a third-party address, and then transfers 3rd amount from global variable from 1st input address to a third-party address.*

6 EVALUATION

We have implemented *Tx2TXT* in 1,500 lines of Python code. Our graph generation tool is built on top of Octopus [1]'s static analysis engine, and our node classification uses Deep Graph Library [5]. We further apply it to real-world smart contracts to evaluate its correctness, effectiveness and usability.

6.1 Experimental Setup

First, to assess the security awareness of our funds-transfer-based graph models, we collect benchmark smart contracts from a state-of-the-art project VeriSmart [56], where 412 contract programs have been confirmed to contain security problems and thus can be used as well-labeled ground truth.

Second, to comparatively check human understanding of our generated descriptions, we create descriptive scripts for 906 contracts **whose developer-crafted descriptions are available** on their DApp websites or GitHub. Among these, 300 are benign contracts from top Etherscan apps, 196 are malicious from HoneyBadger [58], and 412 are vulnerable ones from VeriSmart [56].

Last but not least, to evaluate our machine learning model, we generate 6,000 FTGs from top 2,400 open-sourced contracts in Etherscan [3] (with the highest amounts of transactions), as well as 573 C and Android logic bomb programs whose crucial conditions are labeled. We use 5,400 graphs as the training samples and 600 as the testing samples. Our experiments have been conducted on a server equipped with Intel Xeon Gold 6330 CPU @ 2.00GHz and 256GB memory. The OS is Ubuntu 20.04 LTS (64bit).

6.2 Correctness and Security of FTGs

First, we would like to evaluate our graph models. Particularly, we expect to see *(a)* if our graphs are complete and precise, and *(b)* whether our "transfer"-oriented graph models are security-aware – i.e., whether our graphs can actually capture all the potential security risks in smart contract code.

Accuracy. We manually verify the the completeness and precision of the 780 FTGs generated from the 412 benchmark contracts. In theory, false positives in our dataflow analysis may originate from how we address the memory model in Solidity. Particularly, when we identify the "free memory pointer", we only look for the constant offsets and do not handle the access to this pointer via computed variables. Nevertheless, our results show that *Tx2TXT* does not yield any false negatives in practice. This may be due to the fact that the contracts in this dataset do not contain very complex or unconventional storage/memory accesses. In contrast, we do observe a false positive rate of 3.64%. 224 out of 6160 total nodes are imprecisely added into our graphs. Our further investigation indicates that these false positives are caused due to our conservative way to handle

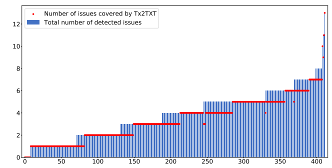


Figure 10: The Coverage of Security Risks

aggregate data types such as arrays. For instance, in a *Lottery* contract [10], while the reward is transferred to the only winner, the winner's address is stored in an array of all players. As a result, any write to any array element is further tracked.

Security Awareness. To evaluate the security awareness of our graph models, we apply multiple security analyzers to the source code of the benchmark contracts and collect their detection results. Particularly, we have used four different tools including VeriSmart [49], SmarTest [48], Slither [2] and Oyente [38]. We count the total number of unique security problems and identify the specific instructions containing these problems. Then, we generate the FTGs of these contracts and check how many detected problems can be covered by our graph models.

Figure 10 illustrates the comparison results, where the blue bars represent the total numbers of detected security issues for each contract while the red dots denote how many are covered by our graphs. As you can see, our "transfer-oriented" graphs can capture a large majority of security issues (96.8% on average). This confirms that security threats in smart contracts indeed lies in their insecure or incorrect funds transfers.

Our graph model does miss a small number of suspicious contract activities because their host functions do not involve funds transfers. In fact, these "security risks" may not cause direct financial damage to end users – despite the discovered suicidal or self-destructive operations in their code, the host contracts do not provide any interfaces for users to *transfer* funds to the contracts in the first place.

6.3 Readability and Understandability

To assess the usability of our generated descriptions, we perform a user study using Amazon Mechanical Turk (MTurk) [17]. We aim to evaluate *(a)* whether human readers can understand our machine-generated texts, and *(b)* whether end users can correctly avoid using risky contracts after they have read our security-centric descriptions.

It is worth noting that it is in fact a very challenging task to conduct effective user studies. While how to avoid biased survey settings and results by itself is an interesting research topic, it is not the major focus of this work. Here, we just follow prior user studies [29, 63] on human understanding of security-related texts to implement our experiments. *Methodology.* We present different types of smart contract textual descriptions to human readers and measure their reaction. Particularly, we collect developers' descriptions (Condition 1.1, 2.1, 2.4, 2.7), security analyzer reports (Condition 2.2, 2.5, 2.8), and *Tx2TXT* descriptions (Condition 1.2, 2.3, 2.6, 2.9). The details of these conditions are in hypothesis 1 and hypothesis 2. Here, *security reports* are textual reports from HoneyBadger and Securify plus the developers' descriptions. *Tx2TXT* descriptions are our generated descriptions plus the reports from HoneyBadger and Securify. We choose to use these two analyzers for this experiment because *(1)* they can particularly analyze Solidity bytecode, *(2)* they generate textual analysis results, and *(3)* they each focus on an individual aspect of security problems (i.e., malicious and vulnerable, respectively). An example of descriptions can be found in online appendix B [20].

Dataset. We perform the user study based upon the descriptions of 890 benign, malicious and vulnerable contracts. For readability study, in order to obtain sufficient responses for each contract, we randomly select 60 contracts from this dataset. For security analysis, we use the entire dataset.

Recruitment of Participants. We recruit participants directly from MTurk and we require participants to have basic knowledge about blockchain and smart contracts. We therefore ask screening questions to ensure participants can correctly identify a smart contract as "a computer program running atop a blockchain" rather than for example "a supplementary contract" used in life insurance. Our study has received an IRB waiver from each author's institution. Besides, we did not collect any sensitive or personal information about participants.

Hypotheses and Conditions. **Hypothesis 1: machine-generated contract descriptions are readable to human readers that have basic knowledge about smart contracts.** To assess the readability, we prepare the developers' descriptions (Condition 1.1) and *Tx2TXT* descriptions (Condition 1.2). We use task-based studies to evaluate how well machine-generated texts are understood by human readers.

Hypothesis 2: Funds transfer-based security-centric descriptions can help reduce the adoption of risky contracts. To assess the effectiveness of *Tx2TXT* descriptions, we present the *Developers'* human-crafted descriptions, the *Security Reports* and *Tx2TXT* descriptions for vulnerable (Condition 2.1, 2.2, 2.3), malicious (Condition 2.4, 2.5, 2.6), and safe (Condition 2.7, 2.8, 2.9) contract functions. We expect to assess the contract adoption rates for individual descriptions on different conditions.

Deployment of User Study. We conduct a within-subjects study. Particularly, we post all the descriptions on MTurk and anonymize their sources. We inform the participants that the tasks are about smart contract descriptions and we pay 0.1 dollars for each task (i.e., completing all survey questions). Participants take part in two sets of experiments.

Readability. First, each participant is given a mixture of 16 descriptions randomly selected from two categories (*Developers'* and *Tx2TXT*'s). After reading each description, they are asked to answer a multiple choice question to check whether they can grasp the meaning of the descriptive sentences. For instance, in the stem, we can present a description "*The function calculates an amount_0 using a timestamp, and transfers the amount_0 from a user input address to an address from global variable.*", and ask a question: "*What is the amount of funds that has been transferred?*". Then, we provide four options "*(1) amount_0*", "*(2) a user input address*", "*(3) an address from global variable*" and "*(4) Not applicable*". If a participant can choose the correct one "*(1) amount_0*", we consider that she can understand the description.

Effectiveness. Second, we present the participants another random sequence of 16 descriptions. One sequence can contain three types of descriptions: *Developers'*, *Security Report* and *Tx2TXT*. Participants are first presented with the expected functionality of a financial application. For example, we inform human readers of what they can expect from an online gambling game: "If you are the winner, the contract must transfer the jackpot (all of its accumulated balance) to your account, and must not transfer any part of it to other accounts." Note that, in real-world use scenarios, this contextual information is not necessary as users of specific contracts must have a general understanding of the application logic (e.g., English/Dutch auctions, election, gambling). However, our participants do not have access to *concrete* smart contract applications during this survey, and thus such baseline knowledge is required for them to interpret the correct application logic and to identify any deviation from the baseline in given descriptions. Once participants have learned the context, they will then be asked to read the descriptions of a "specific implementation" of the aforementioned application and answer a question: "*Do you think this is a secure and fair application that you will use?*" We particularly point out "secure and fair" in order to avoid responses due to any other factors.

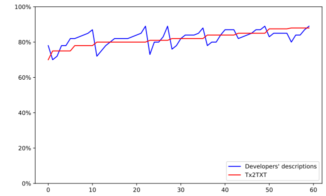


Figure 11: Readability of Descriptions

We have also deployed a validity test to each questionnaire. Particularly, we add two simple attention questions to each questionnaire in order to check whether participants have made sufficient efforts to read and comprehend the given texts. We exclude the responses that do not pass this test.

Results and Implications. **Readability.** We receive 152 valid responses and in total 2432 answers to our readability tasks. Figure 11 illustrates the ratio of correct solutions for every contract on Condition 1.1 (*Developers'*) and 1.2 (*Tx2TXT*). The x-axis is the contract ID while the y-axis is the correctness rate (readability). The two curves represent the results obtained on the two conditions. As you can see, the correctness rate for the *Tx2TXT* descriptions (red curve) is comparable to that of the human-crafted natural language scripts. While the average readability for developers' descriptions is 83.6%, *Tx2TXT* reaches an average score of 82%. This indicates that our machine generated descriptions can successfully be interpreted by human readers.

Nevertheless, we do observe that certain descriptions produced by *Tx2TXT* yields a relatively low readability score (around 70%). For instance, when participants read this description: "The function calculates an amount0 using a timestamp and a global variable, and then transfers this amount from a user input address to the contract address, and then calculates another amount1 using amount0, and then transfers amount1 to a user input address", 30% of readers mistakenly believe that the "amounts of funds that have been transferred" are *timestamp* and *global variable* rather than *amount0* and *amount1*. In this case, although their answers are incorrect, they are still relevant to the provided sentence and thus do not necessarily imply that readers have totally misunderstood the text.

Effectiveness. Table 2 depicts the likelihood that the participants will still choose to use a contract after they have read its descriptions. Specifically, we have received 686 valid responses to our questions.

In general, *Developers'* descriptions are not security sensitive. Regardless whether a contract is risky, a large majority of users (up

Table 2: Contract Adoption Rate

#	Condition	Rate
2.1	Vulnerable w/ *Developers*	79.5%
2.2	Vulnerable w/ *Security Report*	39.4%
2.3	Vulnerable w/ *Tx2TXT*	30.2%
2.4	Malicious w/ *Developers*	86.3%
2.5	Malicious w/ *Security Report*	30.2%
2.6	Malicious w/ *Tx2TXT*	20.4%
2.7	Safe w/ *Developers*	85.0%
2.8	Safe w/ *Security Report*	83.4%
2.9	Safe w/ *Tx2TXT*	80.2%

to 86.3%) is still willing to use the contract after they have read the descriptive texts provided by developers.

In contrast, both *Security Report* (SECURIFY + HONEYBADGER + developers') and our *Tx2TXT* descriptions can raise users' security awareness, while ours further outperforms the former considerably.

For vulnerable contracts, where timestamps are incorrectly used or reentrancy bugs are present, *Tx2TXT* causes 9.2% more users to stop using these unsafe functions, compared to the security reports. For malicious contracts such as theft of funds, our description can help 9.8% more users avoid the hidden threats. Since both *Tx2TXT* descriptions and *Security Reports* contain detection results from SECURIFY and HONEYBADGER, these increased numbers indicate that explaining clearly how funds transfers are conducted in an insecure contract, in addition to abstract analysis reports, is very useful for human users to understand and thus avoid risks.

In the meantime, *Tx2TXT* does not significantly affect the adoption rate of safe contracts and therefore does not cause serious usability issues. This is because our descriptions are faithful to the intrinsic funds flows of target contracts, and thus are consistent with users' expectation for normal application logic.

6.4 Classification of Condition Nodes

We further evaluate the usefulness of our node classification. We hope to check *(a)* whether our trained model can completely identify security-sensitive conditional statements and *(b)* whether the number of selectively added nodes is relatively small.

Accuracy. In the 600 testing samples, we do not observe any false negatives; all 62 security-sensitive nodes can be correctly classified. Besides, our classifier only causes false positives in 1.3% cases. Indeed, the false positive rates for these misclassified cases can be relatively high and may be sometimes over 30%. However, those cases often have a small number of conditional statements, and therefore misclassifying even one or two nodes can result in high FP rates. Note that, again, our selection of condition nodes is designed to be *safe* as we do not want to miss any security-related conditions. In contrast, accidentally preserving less interesting nodes may still be acceptable as long as the generated descriptions are human-readable.

Effectiveness. To evaluate the effectiveness, we compare the total number of condition nodes and the number of selected condition nodes. Overall, the selected nodes merely amount to 4.8% of total conditions. In fact, only 7.23% of the contracts contain security-related conditions that need to be added to `FTG`s. For this 7.23%, on average, the number of selected nodes takes 46% of the total conditions. This high ratio is due to the small amount of conditional statements in these functions where at most four conditions are used. Our case study can be found in online appendix C [20].

6.5 Runtime Performance

Overall, *Tx2TXT* is efficient. Our graph construction and description generation are fast. On average, it takes 1.09 seconds to generate one `FTG` and 0.2 seconds to translate a graph to texts. Our GCN model training costs 5.7 minutes while the testing phase for each graph takes only 10 seconds.

7 RELATED WORK

Verifying the Safety of Smart Contracts. Prior efforts [31, 33, 35, 37, 38, 44, 46, 47, 49, 59, 60] have been made to automatically verify smart contract code so as to detect safety problems. While some aimed to discover syntax based low-level errors, such as transaction-ordering dependence, timestamp dependence [38], flawed bytecode instructions [37], callback-based reentry vulnerabilities [33, 46] and inter-contract vulnerability analysis [31], more recent studies [35, 44, 48, 49, 59] have started to investigate the semantic-level defects that can cause fairness issues.

Correlating Descriptive Text to Program Behaviors. Studies have tried to correlate texts to sensitive behaviors, such as permissions in Android [34, 42, 45] and security related functionalities in IoT [57]. WHYPER [42] used NLP technique to identify sentences that describe the need for a given permission. AutoCog [45] developed a learning-based algorithm to automatically derive a model that correlates textual descriptions with Android permissions. AsDroid [34] further inferred the semantics of the text on those widgets that are associated with the top level functions. SmartAuth [57] combined NLP and program analysis to distill the contextual semantics of IoT apps. Unlike these studies that leverage unique APIs to infer program semantics, *Tx2TXT* proposes a novel semantic model to handle smart contract code.

Software Description Generation. Many efforts [24, 39, 52–54] have been made to generate software descriptions for legacy Java programs. Several previous studies [63, 66] have also attempted to expose security risks in textual descriptions. However, they heavily rely on the unique application semantics provided by the Android framework. Recent work [64] has been done to summarize smart contract functions based on developers' comments, while *Tx2TXT* aims to directly capture program logic from code.

8 CONCLUSION

We develop *Tx2TXT* to automatically create security-centric textual descriptions from smart contract bytecode. We formally define a *funds transfer graph* to model critical funds flows in smart contracts, and employ a *GCN*-based model to identify security-related conditions and selectively add them to our graph models. Our results have shown that *Tx2TXT* outperforms state-of-the-art solutions and can effectively help end users avoid risky contracts.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their constructive comments. 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] 2020. Octopus: Security Analysis tool for Blockchain Smart Contracts. https://github.com/quoscient/octopus/. (2020).

[2] 2021. Slither, the Solidity source analyzer. https://github.com/crytic/slither. (2021).

[3] 2021. The Ethereum Blockchain Explorer. https://etherscan.io/. (2021).

[4] 2022. Compound. (2022). https://compound.finance/

[5] 2022. Deep Graph Library. https://www.dgl.ai/. (2022).

[6] 2022. Discover, collect, and sell extraordinary NFTs. (2022). https://opensea.io/

[7] 2022. Ethereum Dapps Ranking. (2022). https://www.dapp.com/dapps/ethereum-marketplace?sort=4&time=0&type=0

[8] 2022. EY Blockchain Analyzer: Smart Contract & Token Review. (2022). https://www.ey.com/en_us/blockchain-platforms/smart-contract-token-review

[9] 2022. Oyente. (2022). https://github.com/enzymefinance/oyente

[10] 2022. react-ethereum-lottery. https://github.com/omarbastos/react-ethereum-lottery/blob/f519d5baaaad7b8e26c36bca1f45d7cb69bfa2d3/src/ethereum/\contracts/Lottery.sol. (2022).

[11] 2022. Smart contract security service for Ethereum. (2022). https://mythx.io/

[12] 2022. Software Assurance. (2022). https://www.trailofbits.com/services/software-assurance

[13] 2022. Trade NFTs, Get Rewards. (2022). https://looksrare.org/

[14] 2022. TriggerZoo. https://github.com/JordanSamhi/TriggerZoo. (2022).

[15] 2022. TSOpen. https://github.com/JordanSamhi/TSOpen. (2022).

[16] 2022. UNISWAP PROTOCOL. (2022). https://uniswap.org/

[17] 2023. amazon mechanical turk. https://www.mturk.com/mturk/welcome. (2023).

[18] 2023. Ethereum Signature Database. https://www.4byte.directory/signatures/. (2023).

[19] 2023. Function Selector. https://solidity-by-example.org/function-selector/. (2023).

[20] 2023. Online Appendix. (2023). https://github.com/Pypy233/Tx2TXT/blob/master/appendix.pdf

[21] 2023. Securify v2.0. (2023). https://github.com/eth-sri/securify2

[22] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 468–471.

[23] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems* 26 (2013).

[24] Raymond P.L. Buse and Westley R. Weimer. 2010. Automatically Documenting Program Changes. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*.

[25] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, Yuxing Tang, Xiaodong Lin, and Xiaosong Zhang. 2021. SODA: A Generic Online Detection Framework for Smart Contracts. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021*.

[26] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*.

[27] Wei Chen, David Aspinall, Andrew D. Gordon, Charles Sutton, and Igor Muttik. 2016. A Text-Mining Approach to Explain Unwanted Behaviours. In *Proceedings of the 9th European Workshop on System Security (EuroSec '16)*.

[28] Michael del Castillo. 2016. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft. https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft. (2016).

[29] Adrienne Porter Felt, Robert W. Reeder, Hazim Almuhimedi, and Sunny Consolvo. 2014. Experimenting at Scale with Google Chrome's SSL Warning. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'14)*.

[30] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 480–491.

[31] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. {ETHBMC}: A Bounded Model Checker for Smart Contracts. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[32] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. 2016. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In *2016 IEEE Symposium on Security and Privacy (Oakland)*.

[33] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. 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)*.

[34] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction *(ICSE 2014)*.

[35] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings of the 2018 Network and Distributed System Security Symposium*.

[36] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[37] Johannes Krupp and Christian Rossow. 2018. TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium (USENIX Security'18)*.

[38] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*.

[39] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. 2013. Automatic Generation of Natural Language Summaries for Java Classes. In *Proceedings of the 2013 IEEE 21th International Conference on Program Comprehension (ICPC'13)*.

[40] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18)*.

[41] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/. (2022).

[42] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22nd USENIX Conference on Security (SEC'13)*.

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

[44] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (Oakland)*.

[45] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the Description-to-Permission Fidelity in Android Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*.

[46] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *Proceedings of the 2019 Network and Distributed System Security Symposium*.

[47] Evgeniy Shishkin. 2019. Debugging Smart Contract's Business Logic Using Symbolic Model Checking. *Programming and Computer Software* 45, 8 (2019), 590–599.

[48] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1361–1378. https://www.usenix.org/conference/usenixsecurity21/presentation/so

[49] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (Oakland)*.

[50] Solidity. 2022. Solidity. https://solidity.readthedocs.io/en/v0.6.1/. (2022).

[51] Solidity. 2023. Conventions in Solidity. https://docs.soliditylang.org/en/v0.8.17/assembly.html. (2023).

[52] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards Automatically Generating Summary Comments for Java Methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*.

[53] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. 2011. Automatically Detecting and Describing High Level Actions Within Methods. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*.

[54] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. 2011. Generating Parameter Comments and Integrating with Method Summaries. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension (ICPC'11)*.

[55] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing, and Baoxu Liu. 2021. Evil Under the Sun: Understanding and Discovering Attacks on Ethereum Decentralized Applications. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1307–1324. https://www.usenix.org/conference/usenixsecurity21/presentation/su

[56] SunBeomSo. 2022. VeriSmart-benchmarks. https://github.com/kupl/VeriSmart-benchmarks. (2022).

[57] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, XianZheng Guo, and Patrick Tague. 2017. Smartauth: User-Centered Authorization for the Internet of Things. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*.

[58] Christof Ferreira Torres, Mathis Steichen, and Radu State. 2019. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA,

1591–1607. https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira

[59] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*.

[60] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. 2018. Formal specification and verification of smart contracts for Azure blockchain. *arXiv preprint arXiv:1812.08829* (2018).

[61] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael R Lyu. 2018. Benchmarking the capability of symbolic execution tools with logic bombs. *IEEE Transactions on Dependable and Secure Computing* 17, 6 (2018), 1243–1256.

[62] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 363–376.

[63] Shao Yang, Yuehan Wang, Yuan Yao, Haoyu Wang, Yanfang Fanny Ye, and Xusheng Xiao. 2022. DescribeCtx: Context-Aware Description Synthesis for Sensitive Behaviors in Mobile Apps. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*.

[64] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. 2021. A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts. In *Proceedings of the International Conference on Program Comprehension (ICPC'21)*.

[65] Le Yu, Tao Zhang, Xiapu Luo, Lei Xue, and Henry Chang. 2016. Toward automatically generating privacy policy for android apps. *IEEE Transactions on Information Forensics and Security* 12, 4 (2016), 865–880.

[66] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. 2015. Towards Automatic Generation of Security-Centric Descriptions for Android Apps. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*.