MANDO-HGT: Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection

Hoang H. Nguyen*  
L3S Research Center  
Leibniz Universität Hannover  
Hannover, Germany  
ehoang@l3s.de

Nhat-Minh Nguyen*  
Singapore Management University  
Singapore  
nmnguyen@smu.edu.sg

Chunyao Xie  
L3S Research Center  
Leibniz Universität Hannover  
Hannover, Germany  
xie@l3s.de

Zahra Ahmadi  
L3S Research Center  
Leibniz Universität Hannover  
Hannover, Germany  
ahmadi@l3s.de

Daniel Kudenko  
L3S Research Center  
Leibniz Universität Hannover  
Hannover, Germany  
kudenko@l3s.de

Thanh-Nam Doan  
Independent Researcher  
Atlanta, Georgia, USA  
me@tndoan.com

Lingxiao Jiang  
Singapore Management University  
Singapore  
lxjiang@smu.edu.sg

Abstract—Smart contracts in blockchains have been increasingly used for high-value business applications. It is essential to check smart contracts’ reliability before and after deployment. Although various program analysis and deep learning techniques have been proposed to detect vulnerabilities in either Ethereum smart contract source code or bytecode, their detection accuracy and scalability are still limited. This paper presents a novel framework named MANDO-HGT for detecting smart contract vulnerabilities. Given Ethereum smart contracts, either in source code or bytecode form, and vulnerable or clean, MANDO-HGT custom-builds heterogeneous contract graphs (HCGs) to represent control-flow and/or function-call information of the code. It then adapts heterogeneous graph transformers (HGTs) with customized meta relations for graph nodes and edges to learn their embeddings and train classifiers for detecting various vulnerability types in the nodes and graphs of the contracts more accurately. We have collected more than 55K Ethereum smart contracts from various data sources and verified the labels for 423 buggy and 2,742 clean contracts to evaluate MANDO-HGT. Our empirical results show that MANDO-HGT can significantly improve the detection accuracy of other state-of-the-art vulnerability detection techniques that are based on either machine learning or conventional analysis techniques. The accuracy improvements in terms of F1-score range from 0.7% to more than 76% at either the coarse-grained contract level or the fine-grained line level for various vulnerability types in either source code or bytecode. Our method is general and can be retrained easily for different vulnerability types without the need for manually defined vulnerability patterns.

Index Terms—vulnerability detection, smart contracts, source code, bytecode, heterogeneous graph learning, graph transformer

I. INTRODUCTION

Smart contracts on blockchain systems have been used for many application domains [1], such as finance, e-commerce, healthcare, logistics, and law. Any bug or security vulnerability in a smart contract deployed in a blockchain can have devastating consequences for both the developers and the users of the smart contracts [2], [3]. Therefore, there is a high demand for various kinds of security assurance techniques for smart contracts, especially for vulnerability detection.

Many studies have been carried out on vulnerability detection in smart contracts based on conventional software testing, analysis, verification techniques [4]–[11]. Such techniques often require certain types of oracles or specifications of the (un)expected patterns or semantics of smart contract code for analysis. Unfortunately, specifying the patterns can take much manual effort and make it troublesome to adapt the tools to the evolving contract languages and types of vulnerabilities. Also, computational complexity makes it very expensive to repeatedly run the techniques on a large set of smart contracts to search for new types of vulnerabilities. Hence, a new class of vulnerability detection techniques has been proposed based on machine learning and deep learning techniques [12]–[17]. Such techniques aim to encode various syntactic and semantic code information via syntax trees, control-flow graphs, or program dependency graphs, among others, and to train automated classifiers to distinguish vulnerable code from normal ones. The learning-based techniques reduce the need for manually specified patterns or specifications, easier to be adapted to new types of code and vulnerabilities as long as some training data is provided. However, existing code learning techniques often treat most nodes and edges homogeneously, ignoring fine-grained differences in the nodes and edges types and their exact locations in the code’s trees

∗ The first author and the second author contributed equally to this work.

H.H.N. conceived the original idea and experimental settings and directed the project. H.H.N. and N.M.N. developed the framework. N.M.N. performed the experiments for the evaluation.
and graphs. This leads to insufficient learning accuracy, and this limitation becomes more pronounced for smart contracts bytecode without source code. Since the structures representing different bytecode become more similar to each other if the types of the specific bytecode instructions are ignored, making it harder to identify vulnerable/bug patterns.

Combining the advantages of previous techniques and progresses in heterogeneous graph learning [18]–[21], this paper aims to develop a new framework for smart contract vulnerability detection, applicable to both source code and bytecode. The main idea of our framework is two-folded:

- First, we represent the contract code, either source code or bytecode, as customized heterogeneous contract graphs (HCGs) that represent control flows and call relations of the code. With the combined representations, we aim to capture the code’s syntactic and semantic information more comprehensively to facilitate learning of code patterns and distinguishing vulnerable code from clean ones.
- Second, we extend the heterogeneous graph transformer (HGT) [21] techniques to learn different types of nodes and edges of the contract graphs and encode the semantics of the code more accurately. The encodings can then be used to train classifiers to recognize vulnerable code.

We name our framework MANDO-HGT, following a previous work [22] that only works for source code and uses a different graph learning technique. Our framework aims to be more general than previous techniques, applicable for either source code or bytecode, can be instantiated with various graph learning techniques, and can be re-trained for new types of vulnerabilities and detect vulnerabilities in large sets of contracts efficiently and accurately. The general approach is useful since it is not uncommon for smart contracts to be deployed without their source code or for the source code to be lost or deleted over time and the approach should be able to handle variation in generated bytecode to some extent when retrained with bytecode variants.

We have curated 55k Ethereum smart contracts from various data sources, including SmartBugs [23], [24] and SolidiFI-Benchmark [25], then verified the labels for 423 buggy and 2,742 clean contracts and evaluated MANDO-HGT on the mixed dataset. Our evaluation shows that MANDO-HGT significantly improves F1-score over other vulnerability detection techniques: (1) Compared to other best-performing learning-based techniques, it improves their F1-score by 0.74% to 22.56% at the contract level and 3.51% to 7.48% at a more fine-grained line level for various vulnerability types in either source code or bytecode; (2) Compared to best-performing conventional analysis-based techniques that detect vulnerabilities at the fine-grained line level, it improves their F1-score by 18.18% to 76.89%; Furthermore, MANDO-HGT can be re-trained to detect different types of vulnerabilities without manually defining bug patterns needed by analysis-based techniques.

We also show that, through a few case studies assisted by recent neural network interpretation techniques [26], [27], the detection results of MANDO-HGT are often meaningful, reflecting our understanding of the bug patterns. We also provide possible explanations for the failures in a few cases where the detection results are wrong, which may guide future improvements to learning-based techniques.

The rest of the paper is organized as follows. Section II provides a motivating example to show the benefits of our method. Using heterogeneous graph transformers together with heterogeneous contract graphs for smart contracts. Section III briefly discusses our differences from related studies. Section IV provides more details of MANDO-HGT. Section V presents our evaluation results and a few case studies on their interpretations. Section VI concludes with future work.

II. MOTIVATION

Motivating Sample Source Code. Figure 1 (Part A) presents a snippet of a smart contract written in Solidity with a reentrance vulnerability. Part B presents the call graph (CG) of the contract, and part C presents a partial sample control-flow graph (CFG) for the `collect` function of the sample contract. Line 11. `msg.sender.call`, is the root cause of the vulnerability of this sample code. collect can be repeatedly called before `balances` is deducted at Line 12, allowing `msg.sender` to receive more values than what is specified by `_am`. In order to catch this so-called reentrance vulnerability, the control-flow and call relations among `msg.sender`, `balances`, and `_am` should be considered.

Motivating Sample Bytecode. Figure 2 shows a snippet of a smart contract written in Solidity, together with its runtime bytecode and the control-flow graph of the bytecode. It contains an access control vulnerability on lines 3–4 as selfdestruct is a critical function in Ethereum, leading to the self-destruction of the smart contract but inadequately protected. Thus, malicious parties can destruct the contract due to missing access controls. This vulnerability would be represented by two node-edge-type relations in our heterogeneous control-flow graph as `DISPATCHER` → `DISPATCHER` and `DISPATCHER` → `LEAF`.

Objectives. Our primary objective is to automatically capture vulnerabilities in either contract source code like Figure 1 and contract bytecode like Figure 2 via our graph embedding techniques. More specifically, our objective is to: (1) Represent the Solidity source code or EVM bytecode as heterogeneous call and control-flow graphs like the example flow charts; (2) Learn the embeddings of the nodes and graphs; and (3) Efficiently and accurately identify if a contract contains vulnerabilities and locate them if its source code is available.

In cybersecurity contexts, vulnerabilities mean special kinds of bugs that can be exploited and cause major security concerns. According to https://dasp.co/, all the bug types mentioned in this work can be vulnerabilities, although indeed, only a few instances of the bug types can be exploited. In this paper, we do not need to handle the differentiation and simply treat them as synonyms.

2When a contract in Solidity source code is compiled, the produced bytecode has two types: creation bytecode is the constructor code of the contract that performs initializations and deploys the runtime bytecode to the blockchain; the constructor code is then discarded, not stored in blockchain.
graph-learning components are independent of the languages our learning-based approach also requires customized front-end parsing and control-flow graph constructions, the graph-learning components are independent of the languages and can be applicable to new vulnerability types.

Learning-based bug detection techniques. There are also many studies based on machine learning and deep learning for detecting bugs in either source, bytecode, or binary. Some studies consider different kinds of code representations and learning techniques for source code in various languages [45]–[53], but very few consider heterogeneous graph learning. A recent study uses heterogeneous graphs for source code [54], but it has not yet been applied to control-flow graphs of smart contracts. Also, among the limited literature on deep learning-based vulnerability detection methods in smart contracts bytecode [13], [16], [55]–[60], some use control-flow and data-flow graphs (and bytecode instruction sequences). However, they still use homogeneous graph learning techniques, while our approach customizes heterogeneous graph learning for both source code and bytecode of smart contracts.

There are other general code representation learning studies [61]–[66] that are different from ours but can potentially be combined with ours to improve its accuracy in encoding bugs. We leave such interesting exploration for future work.

III. RELATED WORK

We discuss the main differences between our work and closely related work on smart contract bug detection.

Conventional bug detection techniques. Many studies on vulnerability detection are based on conventional techniques, such as testing/fuzzing [4], [28]–[31], symbolic execution [5], [6], [32]–[35], static/dynamic program analysis [7]–[9], [36]–[39], and formal verification [10], [11], [40]–[44]. They often need customized implementation of the testing, analysis, and verification algorithms for the specific smart contract language and vulnerability types; their analysis algorithms can be very different for source code and bytecode, limiting their flexibility for new languages or vulnerability types. Although our learning-based approach also requires customized front-end code parsing and control-flow graph constructions, the graph-learning components are independent of the languages and can be applicable to new vulnerability types.

Learning-based bug detection techniques. There are also many studies based on machine learning and deep learning for detecting bugs in either source, bytecode, or binary. Some studies consider different kinds of code representations and learning techniques for source code in various languages [45]–[53], but very few consider heterogeneous graph learning. A recent study uses heterogeneous graphs for source code [54], but it has not yet been applied to control-flow graphs of smart contracts. Also, among the limited literature on deep learning-based vulnerability detection methods in smart contracts bytecode [13], [16], [55]–[60], some use control-flow and data-flow graphs (and bytecode instruction sequences). However, they still use homogeneous graph learning techniques, while our approach customizes heterogeneous graph learning for both source code and bytecode of smart contracts.

There are other general code representation learning studies [61]–[66] that are different from ours but can potentially be combined with ours to improve its accuracy in encoding bugs. We leave such interesting exploration for future work.

IV. APPROACH

Our proposed framework, MANDO-HGT, consists of five main components in the grey boxes in Figure 3: Heterogeneous Contract Graph Generator, Meta Relations Extractor, Node Features Extractor, MANDO-HGT Graph Neural Network, and Two-Phase Vulnerability Detector. The five components are explained in detail below. The input of MANDO-HGT is either the source code or bytecode of one or more Ethereum smart contracts, and the output is the bug predictions for the input contracts at the contract level (for both source code and bytecode) and the line-level (for source code only).

A. Heterogeneous Contract Graph Generator

Definition IV-A.1 (Heterogeneous Graph). A heterogeneous graph is a directed graph $G = (V, E, \tau, \phi)$, consisting of a
ETHERSOLVE \cite{67}, we define six primary types of nodes representing node types can induce much learning overhead. Based on the Ethereum yellow paper \cite{69}, but using all the opcodes vertices set $V$ and an edge set $E$, $\tau : V \rightarrow A$ is a node-type mapping function and $\phi : E \rightarrow R$ is an edge-type mapping function. $A$ and $R$ denote the sets of node types and edge types, and $|A| \geq 2$ and $|R| \geq 1$.

Smart contract code, either source code or bytecode, is processed by the first component of Figure 3, Heterogeneous Contract Graph Generator, and translated into heterogeneous graphs based on control-flow graphs (CFGs) and/or call graphs (CGs). In MANDO-HGT, we use Slither \cite{7} to analyze source code and EtherSolve \cite{67} to analyze bytecode respectively, to construct basic CFGs and CGs. In contrast to previous studies \cite{12}, \cite{68} that only consider homogeneous forms of control-flow graphs where types of nodes and edges are not utilized, we retain most of the structure and semantics of smart contract code through heterogeneous graphs that preserve various node and edge types. In particular, we convert basic CFGs and CGs into heterogeneous forms, called heterogeneous control-flow graphs and heterogeneous call graphs, and fuse them into heterogeneous contract graphs (HCGs).

**Heterogeneous Control-Flow Graphs (HCFGs).** For input bytecode, a CFG may involve all possible opcodes defined in the Ethereum yellow paper \cite{69}, but using all the opcodes as node types can induce much learning overhead. Based on EtherSolve \cite{67}, we define six primary types of nodes representing important opcode blocks, including $\text{ROOT, BASIC, DISPATCHER, FALLBACK, LEAF, and EXIT}$ in MANDO-HGT: $\text{ROOT}$ and $\text{EXIT}$ represent entry and end blocks, respectively; $\text{DISPATCHER, FALLBACK, and LEAF}$ are $\text{BASIC}$ blocks with some unique characteristics. Specifically, a $\text{BASIC}$ block is a sequence of opcodes executed sequentially between a jump destination ($\text{JUMPDEST}$ opcode) and a jump instruction ($\text{JUMP}$ or $\text{JUMPI}$ opcode). A $\text{DISPATCHER}$ block is a $\text{BASIC}$ block with the last opcode being the return or stop opcode. A $\text{FALLBACK}$ block is a $\text{DISPATCHER}$ block that has no call data, and none of the hashes matches when executed ($\text{REVERT}$ opcode). A $\text{LEAF}$ block has the last opcodes being $\text{REVERT, SELFDESTRUCT, RETURN, INVALID, and STOP}$ and has no successors, which means jumping to the $\text{END}$ block in the CFG. In addition, three edge types are used to describe sequential ($\text{NEXT}$) or branching ($\text{TRUE}$ and $\text{FALSE}$) connections between nodes. Notably, the $\text{JUMPI}$ opcode plays a vital role in a conditional branching structure; we add the edge type $\text{FALSE}$ for a branch that leads to the following opcode block when the branch condition is false, and the $\text{TRUE}$ edge type is for the true branch, which is the argument of the $\text{PUSH}$ opcode interpreted as the destination offset for the $\text{JUMPI}$. Thus, a smart contract can be converted to a heterogeneous control-flow graph (HCFG). Figure 2 shows the generated HCFG for the runtime bytecode of a buggy contract.

For input source code, a CFG may involve many types of statements or lines of code. We use typical statement types as the node types for source code’s HCFGs, such as $\text{ENTRY_POINT, EXPRESSION, NEW_VARIABLE, RETURN, IF, END_IF, IF_LOOP, and END_LOOP}$. Similar to bytecode, three edge types are used to indicate statements’ sequential or branching nature, such as $\text{NEXT, TRUE, and FALSE}$. Figure 1 (Part C) shows a sample HCFG generated for the $\text{Collect}$ function in the $\text{MY_BANK}$ contract.

Due to the capabilities and limitations of the tools for generating CFGs (Slither \cite{7} for Solidity source code and EtherSolve \cite{67} for EVM bytecode), a bytecode’s HCFG represents the control flows throughout an entire smart contract, while a source code’s HCFG is only for one function of a contract. To integrate the HCFGs for all functions of a contract, we also utilize call graphs for source code as explained below.

**Heterogeneous Call Graphs.** A call graph (CG) represents the invocation relations among functions in one or multiple smart contracts. There are two basic forms of calls in smart contracts that the MANDO-HGT framework considers: $\text{internal calls}$ for function calls within the same contract and $\text{external calls}$ for function calls across contracts, represented by two edge types $\text{INTERNAL_CALL}$ and $\text{EXTERNAL_CALL}$ respectively. In addition to the typical function node type $\text{FUNCTION_NAME}$, we also employ the $\text{FALLBACK_NODE}$ node type to represent fallback functions that are executed if a function identifier to be called does not match any accessible function in a smart contract or if insufficient data was provided for the function call. Such fallback functions are directly or indirectly related to numerous Ethereum smart contract vulnerabilities \cite{70}. Figure 1 (Part B) shows such a heterogeneous call graph.

We also use Slither to process each smart contract source code to produce its heterogeneous call graph and add the explicit types to the nodes and edges.

**Heterogeneous Contract Graphs (HCGs): Fusion of Heterogeneous Call Graphs and Heterogeneous Control-Flow Graphs.** The topologies of these two kinds of graphs for a smart contract source code can be merged into a global graph, to facilitate the graph learning process later. In MANDO-HGT, the sub-component Heterogeneous Graphs Fusion is for this purpose: for each node in the heterogeneous call
graph that represents a function, a bridging edge is added to link this node to the entry node of the heterogeneous control-flow graph for the function (as illustrated by the double-arrow edge between Figure 1 Part B and Part C). We call such a fused graph a heterogeneous contract graph (HCG). For bytecode, since the heterogeneous control-flow graph generated by EtherSolve has represented the entire smart contract, our Heterogeneous Graphs Fusion sub-component directly utilizes the bytecode’s HCFG as the fused HCG.

B. Meta Relations Extractor

**Definition IV-B.1 (Meta Relation).** A meta relation of an edge $e = (s, t)$ from a source node $s$ to a target node $t$ is indicated as $\langle \tau(s), \phi(e), \tau(t) \rangle$, with $\tau(s)$ and $\tau(t)$ representing the node type of $s$ and $t$, respectively, and $\phi(e)$ representing for the edge type of $e$. A metapath can refer to a sequence of such meta relations for a sequence of connected edges.

The component Meta Relations Extractor of MANDO-HGT extracts customized meta relations from the generated heterogeneous contract graphs (HCGs). The main advantage of extracting meta relations is avoiding the explosion of all possible node and edge types combinations in the traditional approaches that use metapath [19], [20] as the number of node types and edge types in the graphs is dynamic and can be up to eighteen node types and five edge types.

We also add meta relations through reflective connections between adjacent nodes, e.g., the relation between two adjacent nodes of the types `EXPRESSION` and `END_IF` in Figure 1 can be described by both $\langle \text{EXPRESSION, next, END_IF} \rangle$ and $\langle \text{EXPRESSION, back, END_IF} \rangle$. HCGs are predominantly tree-like, with only a few back-edges created by LOOP-related statements. Adding the reflective relations increases the comprehensiveness of the extracted meta-relations, and improves the stable operability of the heterogeneous graph transformer (HGT) used in MANDO-HGT because the original architecture of each HGT layer requires at least two source nodes for one target node [21] and, without reflective relations, many nodes having only one source node (e.g., the two `EXPRESSION` nodes in Figure 1) would be ignored during training.

C. Node Features Extractor

The main goal of this extractor is to generate basic node features via one of the two following ways. (1) Generate node embeddings via some basic graph neural network without considering node and edge types. Omitting node and edge types is often reasonable for graph classification, as the connectivity topology among nodes is often sufficient to differentiate the graphs. We employ either homogeneous (e.g., node2vec [71]) or heterogeneous (e.g., metapath2vec [19]) graph neural networks in our evaluation (Section V). (2) Generate one-hot vectors based on the node types as the node features. These node features were used as initial embeddings for the MANDO-HGT’s first layer to leverage the rich information independently without relying on any other neural network.

D. MANDO-HGT Graph Neural Network

Figure 4 illustrates the architecture of the MANDO-HGT Graph Neural Network, based on Heterogeneous Graph Transformer (HGT) [21]. In MANDO-HGT GNN, we feed all pairs of meta relations of every target node, including their node types and node features, as inputs to one HGT layer. This is the major difference in our framework from the original HGT GNN. Such a mechanism allows MANDO-HGT to learn the inter-relation among our customized meta relations in HCGs. It is important since it disentangles complex node/edge relations for learning. The outputs of MANDO-HGT GNN are fed into the final components Two-Phase Vulnerability Detector to identify whether the smart contracts contain bugs and to find the bug locations in the contract source code.

**Heterogeneous Graph Transformer (HGT) layer.** The goal of this layer is to learn the attention of every pair of meta relations between a target node $t$ and its neighbor source nodes $s_1$ and $s_2$ [21]. To achieve the goal, the architecture of Transformer [72] is employed with the target node $t$ as the “Query” vector and its neighbors $s_1$ and $s_2$ as “Key” vectors. The attention is the output of the softmax layer applied to the concatenation of the output of $h$ attention head [73]. Each attention head explores a different relation aspect of the two pairs of $t$ with $s_1$ and $t$ with $s_2$ by letting the embedding vectors of $t$ with $s_1$ and $t$ with $s_2$ go through the $l$-th GNN layer denoted by $H^{(l-1)}[t]$, $H^{(l-1)}[s_1]$ and $H^{(l-1)}[s_2]$ in
specifically, there are three sub-components inside the HGT layer: (1) Heterogeneous Mutual Attention: The sub-component uses the $Q$ and $K$ linear transformations of target node $t$, and the two source neighbors $s_1$ and $s_2$ of $t$ as the inputs, and the output is the attention or correlation probability of the two node pairs, i.e., $s_1$ or $s_2$ with $t$ as well as the edge types $\phi(e_1)$ and $\phi(e_2)$ associated with the two given source nodes. The matrix $W^\text{ATT}$ encodes multiple semantic relations of the pairs with the same node type. (2) Heterogeneous Message Passing: The input in this sub-component is the $V$ linear transformations of the pair of source nodes $s_1$ and $s_2$, and the output is the multi-head message containing the distribution differences of nodes and edges with different types. We use matrix $W^\text{MSG}$ to capture the edge dependency of each head. Note that the sub-component does not depend on the one above, so it can be processed simultaneously as the previous one. (3) Target-Specific Heterogeneous Message Aggregation: The sub-component is a multi-layer perception whose input is the aggregation of the outputs of the two components above, and the output is the contextualized representative vector $H(t)$ of node $t$. Also, this sub-component uses an Exponential Linear Unit (ELU) as an activation function.

The target node $t$ goes through $L$ HGT layers to create the embedding vector $H(t)$ such that a mechanism ensures the final embedding vector of $t$ is considered on multiple aspects through transformer architecture.

Optimization for Detection. For graph or node classification tasks, we use a multi-layer perceptron (MLP) with the softmax function as an activation function. The input of the layer depends on the prediction tasks. To reduce the effects of derivative saturation, we use cross entropy as the loss function during training, and the parameters of our model are learned through back-propagation with gradient descent algorithms.

E. Two-Phase Vulnerability Detector

This component comprises two primary phases: Coarse-Grained Detection and Fine-Grained Detection. While the former phase assesses clean versus vulnerable smart contracts at the contract level based on the input bytecode or source code, the latter phase determines the line locations of vulnerabilities in the contract source code. One of our new contributions is detecting vulnerabilities at the line level, whereas earlier learning-based methods [12], [14] only report vulnerabilities at the contract or function level.

1) Phase 1: Coarse-Grained Detection: This phase determines whether a smart contract is vulnerable. We employ the heterogeneous contract graphs and their embeddings for each input smart contract, and we train the MLP (Section IV-D) to predict whether a HCG is clean or vulnerable. The MLP can produce a confidence score for each input graph with respect to each bug type; the contract is classified/predicted as buggy when the confidence score for the graph with respect to a bug type is greater than 0.5. This classification helps reduce the search space by filtering out likely clean smart contracts prior to the second phase of line-level vulnerability detection.

2) Phase 2: Fine-Grained Detection: For the smart contracts identified in the previous phase, the node embeddings of their heterogeneous contract graphs will go through node classification to determine if the nodes may be buggy. Similar to Phase 1, the MLP of the node classification step can produce a confidence score for each node in the input graph with respect to each bug type: a node is classified/predicted as buggy when the confidence score for the node with respect to a bug type is greater than 0.5. Note that the nodes correspond to statements or lines in source code, so we can identify the locations of vulnerabilities at the line level in source code 3.

V. Empirical Evaluation

We publicize the datasets and our graph embedding models at https://github.com/MANDO-Project/ge-sc-transformer.

A. Dataset

Our evaluation is carried out on a mixture of three datasets:

(1) Smartbugs Curated [23], [24] is a collection of vulnerable Ethereum smart contracts organized into nine types. It contains 143 annotated contracts having 208 tagged vulnerabilities. (2) SolidiFI-Benchmark [25] is a synthetic dataset of vulnerable smart contracts with 9369 injected vulnerabilities in 350 distinct contracts and seven different vulnerability types. To ensure consistency in the evaluation, we only focus on the seven types of vulnerabilities that are joint in both datasets, including Access Control, Arithmetic, Denial of Service, Front Running, Reentrancy, Time Manipulation, and Unchecked Low Level Calls. Together with Smartbugs Curated, we have 423 buggy smart contracts in total since there are some contracts that could not be processed by Slither [7] or EtherSolve [67]. (3) Clean Smart Contracts from Smartbugs Wild [23], [24] is a set of 47,398 Ethereum smart contracts. We identified 2,742 contracts out of 47,398 that do not contain any bugs using 11 integrated detection tools of Smartbugs. Thus, we use the 2,742 contracts as a set of clean contracts.

The smart contracts in all the datasets are in source code form. We employ Slither [7] to traverse and generate the basic homogeneous forms of CFGs and CGs for the input source code. To get smart contract bytecode, we use Cryptic

TABLE I

<table>
<thead>
<tr>
<th>Bug Types</th>
<th># Total / Buggy Contracts</th>
<th># Total Nodes (source / byte code)</th>
<th># Total Edges (source / byte code)</th>
<th># Buggy Nodes (source code only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Control</td>
<td>114 / 57</td>
<td>13014 / 4447</td>
<td>10721 / 6198</td>
<td>7506</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>120 / 60</td>
<td>17372 / 4796</td>
<td>14271 / 6602</td>
<td>10110</td>
</tr>
<tr>
<td>Denial of Service</td>
<td>92 / 46</td>
<td>13568 / 41066</td>
<td>11997 / 5671</td>
<td>8280</td>
</tr>
<tr>
<td>Front Running</td>
<td>88 / 44</td>
<td>20824 / 51297</td>
<td>16761 / 71652</td>
<td>10008</td>
</tr>
<tr>
<td>Reentrancy</td>
<td>142 / 71</td>
<td>41089 / 46586</td>
<td>17614 / 64798</td>
<td>11238</td>
</tr>
<tr>
<td>Time Manipulation</td>
<td>100 / 50</td>
<td>14785 / 43424</td>
<td>15550 / 60464</td>
<td>10051</td>
</tr>
<tr>
<td>Unchecked Low Level Calls</td>
<td>190 / 95</td>
<td>17756 / 55103</td>
<td>14858 / 75950</td>
<td>7583</td>
</tr>
<tr>
<td>Total</td>
<td>846 / 423</td>
<td>120597 / 338188</td>
<td>119680 / 455491</td>
<td>64770</td>
</tr>
</tbody>
</table>

3 MANDO-HGT can also potentially detect bugs at the instruction level in bytecodes. However, when the bytecode instructions cannot be mapped back to source code lines, the detection results may not be readable by human developers. Thus we do not perform instruction/line-level bug detection for bytecode in this paper.
As a helpful assistant, I've transformed the document into a plain text representation as follows:

### Evaluation

<table>
<thead>
<tr>
<th>Methods</th>
<th>Access Control</th>
<th>Arithmetic</th>
<th>Denial of Service</th>
<th>Front Running</th>
<th>Reentrancy</th>
<th>Time Manipulation</th>
<th>Unchecked Low Level Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Heterogeneous GNN</td>
<td>48.43</td>
<td>54.89</td>
<td>64.13</td>
<td>66.79</td>
<td>64.28</td>
<td>60.73</td>
<td>62.74</td>
</tr>
<tr>
<td></td>
<td>72.80</td>
<td>69.52</td>
<td>69.46</td>
<td>69.86</td>
<td>62.38</td>
<td>64.88</td>
<td>69.23</td>
</tr>
<tr>
<td>Original Homogeneous GNNs</td>
<td>57.22</td>
<td>51.45</td>
<td>61.11</td>
<td>50.74</td>
<td>66.34</td>
<td>65.92</td>
<td>63.79</td>
</tr>
<tr>
<td></td>
<td>68.64</td>
<td>69.20</td>
<td>70.98</td>
<td>69.52</td>
<td>67.39</td>
<td>69.99</td>
<td>70.00</td>
</tr>
<tr>
<td></td>
<td>60.78</td>
<td>61.73</td>
<td>64.16</td>
<td>66.50</td>
<td>62.69</td>
<td>64.53</td>
<td>63.22</td>
</tr>
<tr>
<td></td>
<td>53.64</td>
<td>55.55</td>
<td>49.63</td>
<td>50.27</td>
<td>49.23</td>
<td>50.04</td>
<td>53.15</td>
</tr>
</tbody>
</table>

The best Buggy F1 scores of MANDO:

<table>
<thead>
<tr>
<th>MANDO-HGT with Node Features Generated by</th>
<th>Node Type One Hot Vectors</th>
<th>metapath2vec</th>
<th>LINE</th>
<th>node2vec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>82.86</td>
<td>88.13</td>
<td>89.33</td>
<td>89.30</td>
</tr>
<tr>
<td></td>
<td>79.46</td>
<td>88.93</td>
<td>85.93</td>
<td>87.67</td>
</tr>
<tr>
<td></td>
<td>78.56</td>
<td>88.93</td>
<td>86.69</td>
<td>86.15</td>
</tr>
<tr>
<td></td>
<td>82.75</td>
<td>89.41</td>
<td>89.39</td>
<td>91.25</td>
</tr>
<tr>
<td></td>
<td>77.73</td>
<td>88.39</td>
<td>87.47</td>
<td>86.62</td>
</tr>
<tr>
<td></td>
<td>82.65</td>
<td>87.91</td>
<td>85.86</td>
<td>90.83</td>
</tr>
<tr>
<td></td>
<td>80.67</td>
<td>87.77</td>
<td>87.63</td>
<td>86.19</td>
</tr>
</tbody>
</table>

### Table II

Performance comparison in terms of Buggy-F1 score on different bug detection methods at the contract granularity level for both source code and bytecode. In each cell, the first number is the source code's result, and the second with grey shading is the bytecode's result. We use the Heterogeneous Contract Graphs of both clean and buggy smart contracts as the inputs for MANDO-HGT. The best performance for each bug type is in boldface separately for source code and bytecode. For MANDO, we only report the best performance among node feature generators from their paper [22], due to space limit.

### Discussion

The MANDO-HGT framework is designed to handle both source code and bytecode vulnerabilities. It leverages heterogeneous contract graphs to improve detection accuracy. The framework is flexible in integrating various node feature generators, which is a significant advantage over homogeneous graph neural networks.

**Compiler**: MANDO-HGT uses the Solidity compiler [74] as a foundation, allowing for flexibility in choosing different compilers depending on the version requirements.

**Training Data**: The framework utilizes a variety of training datasets, including Smartbugs, which cover a wide range of vulnerabilities.

**Evaluation**: The evaluation metrics include F1-score and Macro-F1, which are crucial for assessing the performance of bug detection systems. The framework achieves competitive results compared to other state-of-the-art methods.

**Baseline Comparison**: MANDO-HGT outperforms several baselines, including static analysis tools like Mythril [78], Manticore [6], and Oyente [5]. It also compares favorably with other dynamic analysis tools like Seclint, Mythril, and MythrilJS.

### Conclusion

MANDO-HGT provides a comprehensive framework for smart contract vulnerability detection, offering both source code and bytecode analysis capabilities. Its flexibility in handling different compilers and node feature generators makes it a versatile tool for security researchers and developers.

---

**Note**: The exact technical details and results are beyond the scope of this response but are available in the full paper.
### Parameter Settings

All models have their node or graph embedding size set to 128. We employ an adaptive learning rate ranging from 0.0005 to 0.01 for coarse-grained classification and from 0.0002 to 0.005 for fine-grained classification. For each target node and its each meta-relation pair that are fed to the MANDO-HGT GNN, we use two HGT layers \([21]\) and set eight multi-heads whose hidden size is 128. We use their recommended settings for node2vec, LINE, GCN, meta-path2vec, and MANDO to ensure the highest settings.

### D. Experimental Results

In our initial experiments, we divided the 423 buggy contracts and 2,742 clean contracts into the training/validation/test sets using the 60%/20%/20% split ratio. However, some bug types in our dataset have fewer than 50 contracts, resulting in insufficient training/testing samples. In addition, we discovered that the loss value remains constant after a fixed number of epochs (100 and 50 epochs for Fine-Grained and Coarse-Grained tasks, respectively). In order to increase the training/test set sizes and maintain the proportion of vulnerable nodes in each set, we decided to split the contracts into only training/test sets using the 70%/30% ratio for all the bug types. For each setting, embedding method, and bug type, to ensure the robustness of our results we repeated the experiments 20 times with a different random seed and performed t-tests, and set eight multi-heads whose hidden size is 128. We use their recommended settings for node2vec, LINE, GCN, meta-path2vec, and MANDO to ensure the highest settings.

#### 1) Coarse-Grained Contract-Level Vulnerability Detection

Table II shows the average results of 20 independent runs of the baselines and MANDO-HGT at the contract level for both source code and bytecode. We can observe that:

- MANDO-HGT and MANDO with node type generated by one hot vector \([22]\) perform better than other methods, i.e., original heterogeneous and homogeneous GNNs for detecting bugs at source code or bytecode levels. For instance, MANDO-HGT improves up to 34.52% compared to heterogeneous GNN for detecting arithmetic bugs with source code inputs. Moreover, no matter what methods are used to generate node feature, MANDO and MANDO-HGT frameworks still outperforms the baselines.
- Between MANDO-HGT and MANDO, the performance of MANDO-HGT is overall higher than that of the former for both source code and bytecode. However, the performance gaps between the two frameworks for some bug types are relatively small, e.g., 2.24% in access control bugs in the bytecode. For this reason, we apply a t-test to check if the performance of MANDO-HGT is statistically significantly better than MANDO based on the 20 runs of both models for each bug type. All of the t-values are positive while most p-values are less than 0.05, which implies that the performance of MANDO-HGT for most bug types is statistically significantly better than MANDO.
- It is clear that integrating node features via different graph neural networks inside MANDO-HGT outperforms all other GNN baselines, although it is unclear which node features perform the best. This observation is also applicable to various node features used by previous studies, e.g., MANDO. Hence, we believe that an architecture combining different GNNs is useful for classifying buggy contracts.

#### 2) Fine-Grained Line-Level Vulnerability Detection

Table III shows the performances of MANDO-HGT and other baselines based on source code at the line level.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Access Control</th>
<th>Arithmetic</th>
<th>Denial of Service</th>
<th>Front Running</th>
<th>Reentrancy</th>
<th>Time Manipulation</th>
<th>Unchecked Low Level Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANDO-HGT with Node Features Generated by</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>meta-path2vec</td>
<td>78.99</td>
<td>82.99</td>
<td>76.54</td>
<td>89.13</td>
<td>84.06</td>
<td>89.83</td>
<td>76.05</td>
</tr>
<tr>
<td>GCN</td>
<td>87.76</td>
<td>88.86</td>
<td>83.73</td>
<td>92.96</td>
<td>89.59</td>
<td>92.91</td>
<td>89.47</td>
</tr>
<tr>
<td>LINE</td>
<td>86.58</td>
<td>87.98</td>
<td>83.00</td>
<td>92.17</td>
<td>88.77</td>
<td>93.26</td>
<td>90.89</td>
</tr>
<tr>
<td>node2vec</td>
<td>84.23</td>
<td>85.66</td>
<td>81.93</td>
<td>90.46</td>
<td>88.48</td>
<td>92.31</td>
<td>87.60</td>
</tr>
</tbody>
</table>

**TABLE III**

Performance comparison in terms of Buggy-F1 score on different bug detection methods based on source code at the line granularity level. The best performance for each bug type is in boldface.
inherent limitations in relying on predefined patterns that are incapable of capturing these vulnerabilities.

- In MANDO-HGT, node features generated by one hot vector are better than other node feature generating methods for most bug types. The only exception is the node features generated by LINE in MANDO-HGT for detecting unchecked low level calls bugs. For example, for the reentrancy bugs, we got the best Buggy-F1, 92.59%, with the node features based on NodeType One Hot vectors. The models with node features generated by other methods are all lower.

E. Case Studies: Interpreting Vulnerability Prediction Results

To shed light on the reasons why MANDO-HGT can produce successful or failed predictions, we aim to identify certain correlations between MANDO-HGT’s prediction results and the actual semantics of smart contract code in this section. We use a post-hoc local model-agnostic interpretable framework, GraphSVX [27], a state-of-the-art method for graph interpretability, based on Shapley value [79] for graph neural networks to interpret the behaviors and confidence of our model’s predictions versus the input smart contract source code. We also evaluated several other XAI methods, including the original SHAP, and found that the results of GraphSVX were better than others since GraphSVX considers more of the structures of graphs while the SHAP only considers individual object embeddings. Shapley values of an individual feature \( j \) is \( \phi(\text{val}(j)) \) providing a proxy to assess the overall significance of the feature \( j \) to an output of a data point by averaging the marginal contribution of the feature across all possible coalitions where the feature presents. In MANDO-HGT, when making a prediction on a focal node, we consider all neighbors \( N \) of this node as the features which could impact the prediction on the focal node. We obtain the Shapley value by:

\[
\phi(\text{val}(j)) = \sum_{S \subseteq \{1, \ldots, N\} \setminus \{j\}} \frac{|S| (N-|S|-1)!}{N!} (\text{val}(S \cup \{j\}) - \text{val}(S)),
\]

where \( S \subseteq \{1, \ldots, N\} \setminus \{j\} \) are the possible coalitions of node \( j \)’s neighbors, and \( \text{val}(S) = \mathbb{E}[f(X) | X_{S} = x_{S}] - \mathbb{E}[f(X)] \) with \( \mathbb{E}[f(X)] \) being average prediction of dataset \( X \).

We extract neighbor nodes of a focal node and compute their marginal contribution towards the prediction for the focal node, and use some random masking strategy [27] to select subsets of them to calculate their Shapley values more efficiently. The Shapley value of a node implies the contribution of the node to the focal node, and a node’s confidence refers to the confidence score of the model’s prediction for the node. A node is considered buggy if its confidence score is greater than 0.5 with respect to a bug type (cf. Section IV-E).

For any given node used as a focal node for examination, we calculate all of its neighbors’ Shapley values to find out their effect on the model’s bug prediction result. Larger Shapley values indicate the neighbor’s higher impact on the focal node. In the following, we explain several cases where MANDO-HGT makes correct or incorrect predictions to further study possible correlations between the Shapley values of a focal node’s neighbors and its meta relations. For instance, a buggy node might belong to several meta relations, and if there is a meta relation that frequently contains nodes of a certain bug type and contributes the highest Shapley value to the focal node, the focal node’s bug prediction confidence might be high. On the contrary, if some meta relations contain clean nodes contributing high Shapley values to the focal node, the bug prediction confidence for the focal node might be low.

From Figure 5 to Figure 8, subgraphs of the heterogeneous contract graphs are shown next to their source code. The nodes are filled with either red or green, indicating either buggy or clean predictions by our model for the nodes. The circle or triangle shapes of the nodes stand for buggy or clean nodes as indicated by the ground-truth labels. The yellow tag for each node contains information on the node type, its corresponding lines of source code, the confidence score of our model prediction (when the confidence is higher than 0.5, the node is predicted as buggy; otherwise, clean), and the Shapley value of that node to the focal node at the end of the arrow chain in each subgraph.

1) True positive cases: Figure 5-(1) shows a code snippet together with a sub-graph of its HCG that contains an access control bug at Line 27 corresponding to the focal node. We saw a meta-relation pair \( \langle \text{FUNCTION NAME}, \text{next}, \text{ENTRY POINT} \rangle \) and \( \langle \text{ENTRY POINT}, \text{next}, \text{EXPRESSION} \rangle \) frequently appear in our samples. In this case, \( \text{FUNCTION NAME} \) and \( \text{ENTRY POINT} \) nodes correspond to the whole PopBonusCode function were predicted as a buggy node and had approximate Shapley values 0.506 and 0.494 contributing to the focal \( \text{EXPRESSION} \) node, which implies that the focal node is likely buggy too. Indeed, MANDO-HGT correctly predicted the focal \( \text{EXPRESSION} \) node as buggy.

Figure 5-(2) illustrates a smart contract that has arithmetic bugs at Lines 15, 18, 21, 24, 27, and
Fig. 6. True negative case of access control sample

Fig. 7. False positive cases of front running and reentrancy samples

30 because the input variables in these functions are not checked for overflow or underflow before the operations are performed. One of the meta-relation pairs, \( \langle \text{FUNCTION} \text{ NAME}, \text{next}, \text{ENTRY POINT} \rangle \) and \( \langle \text{ENTRY POINT}, \text{next}, \text{EXPRESSION} \rangle \), appears three times in Lines 15, 18, and 21. Lines 24, 27, and 30 correspond to another meta-relation pair: \( \langle \text{FUNCTION} \text{ NAME}, \text{next}, \text{ENTRY POINT} \rangle \) and \( \langle \text{ENTRY POINT}, \text{next}, \text{NEW VARIABLE} \rangle \). The \text{FUNCTION} \text{ NAME} and \text{ENTRY POINT} nodes represent an entire function and its entry point, indicating such bugs often happen at the beginning of a function with specific operations. MANDO-HGT is good at recognizing such frequently appearing bug patterns. Correspondingly, the Shapley values of these nodes to the focal buggy \text{EXPRESSION} and \text{NEW VARIABLE} nodes are around 0.5, which implies that relatively high Shapley values might reflect frequently occurring patterns for bug detection.

2) True negative cases: The code in Figure 6 generated a HCG with meta-relation pair \( \langle \text{FUNCTION} \text{ NAME}, \text{next}, \text{ENTRY} \text{ POINT} \rangle \) and \( \langle \text{ENTRY POINT}, \text{next}, \text{RETURN} \rangle \). Its meta relations with \text{EXPRESSION}, \text{ENTRY POINT}, and \text{FUNCTION} \text{ NAME} node types also differ from those of access control bugs (e.g., the one in Figure 5-(1)). Besides, the Shapley values of \text{ENTRY POINT} and \text{FUNCTION} \text{ NAME} nodes of function \text{isComplete} to the focal \text{RETURN} statement, in this case, are 0.5, indicating that the two nodes (which are clean) have relatively significant impact on the prediction for \text{RETURN}. MANDO-HGT correctly predicted the focal node as a clean node (via a very low bug confidence score 0.002).

3) False positive cases: MANDO-HGT may wrongly predict some clean code as buggy. For example, for front running vulnerabilities, which are typically found in a statement that attempts to transfer a high amount for their transaction to be prioritized, they often involve an \text{EXPRESSION} node, similar to the node for the line 22 in Figure 7-(1). However, some clean statements occasionally preceding the buggy line are also \text{EXPRESSION} nodes, such as lines 19 and 20 in function \text{setReward}. The Shapley values from the buggy \text{FUNCTION} \text{ NAME} and \text{ENTRY POINT} nodes are 0.5, which indicated that they have significant impact to that focal node, causing our model to mistakenly classify the focal node at Line 19 as buggy.

Figure 7-(2) illustrates a similar false positive situation for the reentrancy bug type. The \text{NEW VARIABLE} node corresponds to the code Line 18 defining a new variable and has a reentrancy bug in relation to the reduction of relevant \text{credit} to zero at Line 20. The focal \text{EXPRESSION} node at Line 19 was wrongly predicted by MANDO-HGT, likely because it has data dependency with \text{callResult} defined at the buggy line 18. The buggy nodes like \text{FUNCTION} \text{ NAME}, \text{ENTRY POINT}, and \text{NEW VARIABLE} at Line 18 have relatively higher Shapley values of 0.251, 0.251, and 0.169 to the focal node at Line 19 than the Shapley values from the other two clean nodes (IF at Line 15 and \text{NEW VARIABLE} at Line 14), 0.184 and 0.108. This indicates that the buggy nodes have more impact on the focal \text{EXPRESSION} node at Line 19, causing MANDO-HGT to misclassify it as buggy.

4) False negative cases: MANDO-HGT may also miss certain vulnerabilities. For example, Figure 8 shows a reentrancy bug in the focal node \text{ENTRY POINT} involving lines 15, 16, 17, but our model predicts it as a clean node. By considering the meta relations pair \( \langle \text{FUNCTION} \text{ NAME}, \text{next}, \text{FUNCTION} \text{ NAME} \rangle \) and \( \langle \text{FUNCTION} \text{ NAME}, \text{next}, \text{ENTRY POINT} \rangle \) of the focal node, we have the clean \text{FUNCTION} \text{ NAME} node containing lines 46, 47, 48 with 0.5 Shapley value having the greatest impact to the focal \text{ENTRY POINT} node, higher than the 0.131 Shapley value from the buggy \text{FUNCTION} \text{ NAME} node containing lines 15, 16, 17, causing our model to predict the focal node as clean.

F. Limitations and Discussions

We know that the effectiveness of heterogeneous graph transformers relies on node/edge types and meta relations used,
Besides various hyperparameters for training. The graphs used to represent syntactical and semantic information from smart contract source code or bytecode also significantly impact graph learning. Especially for bytecode, whose syntactical structure is flatter with more obscured semantic information than source code, suitable graph representations become more important for effective learning. Also, although our model can detect fine-grained bugs at the instruction level for bytecode as well, it would be better to map the detected bugs in bytecode back to source code lines for better readability when reporting the results to developers. We restrain our tool from reporting instruction-level bug detection results before we have a reliable way to make the results understandable in relation to their source code.

Lacking labeled data is always an issue when applying supervised learning to classification tasks, especially for smart contracts with limited sample buggy code for training. Although Smartbugs and Solidifi datasets are useful, some smart contracts in the datasets were not annotated. Furthermore, the labels annotated in the datasets for some bugs may not always be objective and agreeable by all developers as there can be subjective and more than one interpretation of the root causes of a bug (e.g., bugs due to some missing lines of code); some labels are not fine-grained enough for each line or even each expression in code. Such inconsistent labels may hinder the training of our models. In our experiments, we excluded unlabelled data and manually checked some data samples and corrected a few inconsistent labels, and used balanced data sets for training and testing, to minimize the impact of inaccurate labels or imbalanced data.

In the experiments, some smart contracts could not be processed by Slither [7] or EtherSolve [67]. One of the main limitations of the tools is that they rely on the Soliﬁty compiler to build the control-ﬂow graph of the contracts, but there are issues related to the version compatibility of Soliﬁty or the uses of optimized/obfuscated bytecode. For example, a smart contract written in an older version of Soliﬁty is not supported by the compiler used by Slither or EtherSolve; the control-ﬂow graph may not be built successfully.

VI. CONCLUSION & FUTURE WORK

This paper proposes a new learning-based vulnerability detection framework for Ethereum smart contract source code and bytecode, named MANDO-HGT, using heterogeneous graph transformer (HGT) techniques. In particular, it constructs heterogeneous contract graphs (HCGs) that represent control ﬂow and function call relations of smart contracts, then deﬁnes customized meta relations based on node/edge types in HCGs, adapts HGT models to generate embeddings for nodes and graphs, and uses the embeddings to train classiﬁers to recognize various types of buggy code at the granularity levels corresponding to either individual contracts or individual lines of code. Our evaluation results on a curated smart contract dataset containing labeled vulnerabilities show that MANDO-HGT can signiﬁcantly improve the accuracy of many previous vulnerability detection techniques, including best-performing learning-based and best-performing conventional analysis-based ones. The improvements in terms of the F1-score range from 0.74% to 76.89% for various bug types and detection techniques.

In the future, we will extend heterogeneous contract graphs to be more comprehensive graph representations of Ethereum smart contracts (e.g., data dependencies and contract calls) for both source code and bytecode and combine heterogeneous graph transformers with more graph learning techniques, especially those suitable for few-shot learning and handling inconsistent labels, for more accurate and usable vulnerability detection. It may also be possible and interesting to utilize results from conventional testing, analysis, and veriﬁcation techniques to help better train and improve our approach.

REFERENCES


