Cross-Modality Mutual Learning for Enhancing Smart Contract Vulnerability Detection on Bytecode

Peng Qian
Zhejiang University
Hang Zhou, China
messi.qp711@gmail.com

Zhenguang Liu*
Zhejiang University
Hang Zhou, China
liuzhenguang2008@gmail.com

Yifang Yin
A*STAR
Singapore
yin_yifang@i2r.a-star.edu.sg

Qinming He
Zhejiang University
Hang Zhou, China
hqm@zju.edu.cn

ABSTRACT

Over the past couple of years, smart contracts have been plagued by multifarious vulnerabilities, which have led to catastrophic financial losses. Their security issues, therefore, have drawn intense attention. As countermeasures, a family of tools has been developed to identify vulnerabilities in smart contracts at the source-code level. Unfortunately, only a small fraction of smart contracts is currently open-sourced. Another spectrum of work is presented to deal with pure bytecode, but most such efforts still suffer from relatively low performance due to the inherent difficulty in restoring abundant semantics in the source code from the bytecode.

This paper proposes a novel cross-modality mutual learning framework for enhancing smart contract vulnerability detection on bytecode. Specifically, we engage in two networks, a student network $\mathbb{S}$ as the *primary network* and a teacher network $\mathbb{T}$ as the *auxiliary network*. $\mathbb{T}$ takes two modalities, *i.e.,* source code and its corresponding bytecode as inputs, while $\mathbb{S}$ is fed with only bytecode. By learning from $\mathbb{T}$, $\mathbb{S}$ is trained to infer the missed source code embeddings and combine both modalities to approach precise vulnerability detection. To further facilitate mutual learning between $\mathbb{S}$ and $\mathbb{T}$, we present a cross-modality mutual learning loss and two transfer losses. As a side contribution, we construct and release a labeled smart contract dataset that concerns four types of common vulnerabilities. Experimental results show that our method significantly surpasses state-of-the-art approaches.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Computing methodologies → Knowledge representation and reasoning**.

*Corresponding author.

KEYWORDS

Smart contract; Bug detection; Cross modality; Mutual learning

ACM Reference Format:
Peng Qian, Zhenguang Liu, Yifang Yin, and Qinming He. 2023. Cross-Modality Mutual Learning for Enhancing Smart Contract Vulnerability Detection on Bytecode. In *Proceedings of the ACM Web Conference 2023 (WWW '23), April 30–May 04, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3543507.3583367

1 INTRODUCTION

Blockchain has received considerable attention both in practice and in the research community over the past decade [28, 45]. A blockchain is essentially a distributed and shared ledger maintained by worldwide bookkeeping nodes (*a.k.a* miners), which follow a well-designed consensus protocol that dictates the appending of new blocks [39]. The duplicate ledgers distributedly stored in the bookkeeping nodes enforce transactions immutable, endowing the blockchain with tamper-proof and decentralized nature [7].

Modern blockchains, such as Ethereum [43], enable the execution of *smart contracts*, which are programs running on top of a blockchain system [39]. Developers are capable of implementing arbitrary rules into the smart contract code for controlling digital assets. Once the code is deployed on the blockchain, its defined rules are automatically executed. Owing to the immutability of blockchain, the execution of a smart contract strictly complies with its pre-defined rules (i.e., contract terms) and is unalterable, making it impartial to all stakeholders.

So far, Ethereum reaches a market capitalization of over $178 billion [10]. As it becomes more prevalent and carries more value, attackers become more incentivized to unearth and exploit potential problems in smart contracts. In fact, Ethereum smart contracts have already faced a considerable number of devastating vulnerability attack incidents, which have resulted in substantial economic losses [31]. For example, in 2017, over $150 million worth of Ether (*i.e.,* the cryptocurrency of Ethereum) was frozen due to the *delegatecall* vulnerability [36]. Recently, attackers exploited the *reentrancy* vulnerability in the *Cream.Finance* contract to steal more than $130 million worth of digital assets [18]. Distinct from conventional programs that can be updated when bugs are exposed, there is no way to patch smart contract bugs unless subverting the blockchain (*namely* controlling more than 51% computing power of the whole blockchain network [22]), which is almost impossible. Obviously,

various flaws in smart contracts have become a serious security threat. Effective vulnerability checkers for smart contracts are much coveted, ideally before their deployments to the blockchain.

Existing Methods and Challenges. Upon scrutinizing the released implementations of existing methods, we empirically found that current bug detection approaches for smart contracts can be roughly funneled into two categories. One line of work [19, 40] revolves around conventional static analysis and fuzzing techniques. They leverage fixed patterns to identify vulnerabilities. However, it is *non-trivial* to define perfect patterns for complex vulnerabilities. Another line of effort [24, 52] builds upon the superiority of deep learning in handling sophisticated data, leading to impressive performance gains. It is worth mentioning that current methods tend to detect vulnerabilities from contract source code since it contains complete semantic information. Unfortunately, the vast majority of smart contracts are not open-sourced and only the bytecode can be readily accessed [17]. While existing works [6] can handle bytecode solely, they fail to obtain a high accuracy due to the difficulty in restoring rich control and data flow semantics from the bytecode.

Notwithstanding the significant differences between source code and its compiled bytecode, their intimate connection is undeniable and one form may appear incomplete without the other. In vulnerability detection, these two modalities could complement each other, where we expect the source code to contribute comprehensive control and data flow dependencies while the bytecode contributes succinct instruction information. However, in smart contract vulnerability detection on bytecode, the challenge is that we have only bytecode available while the source code is missing. It is well known that reverting the bytecode back to the source code is extremely difficult, especially for smart contracts [23]. We are ambitious to know whether it is possible to improve smart contract vulnerability detection on bytecode with the assistance of source code even when the source code is missing in the inference phase.

This motivates us to come up with the following three key designs. **(1)** Using the collected 40K smart contracts that have both source code and bytecode, we train a teacher network $\mathbb{T}$ that absorbs two modalities, *viz.,* source code and corresponding bytecode as inputs, and outputs the binary label which indicates whether the tested function has a specific vulnerability. Technically, we cast both source code and bytecode into graph structures and use graph attention networks to handle them. **(2)** We then train a student network $\mathbb{S}$, *namely the primary network*, that distills knowledge from a teacher network $\mathbb{T}$, *namely the auxiliary network*. Specifically, by learning from $\mathbb{T}$, $\mathbb{S}$ has the capacity to predict source code embeddings from bytecode embeddings. In the *inference* phase, $\mathbb{S}$ could infer the missed source code embeddings from the bytecode and combine both modalities to approach accurate detection. **(3)** To further enable mutual learning between $\mathbb{S}$ and $\mathbb{T}$, we propose a cross-modality mutual learning strategy framed with theoretically motivated losses. We would like to highlight that we do not try to reconstruct the entire source code, instead, we infer the source code embeddings that are beneficial to enhance the accuracy of vulnerability detection on bytecode.

Extensive experiments show that our approach achieves significant performance gains over the state-of-the-art: *accuracy* from 81% to 84%, 82% to 90%, 73% to 79%, and 73% to 79% on four types of common vulnerabilities, respectively.

To summarize, we make the following **key contributions**:

- We investigate whether the mutual learning strategy could help in the challenging scenario where smart contract source code is missing. To the best of our knowledge, we are the first to investigate the idea of utilizing mutual learning to enhance smart contract vulnerability detection.
- We propose a novel teacher-student framework for smart contract vulnerability detection on bytecode, achieving effective knowledge transfer from a dual-modality teacher network to a single-modality student network.
- Our approach sets the new state-of-the-art performance and overall provides interesting insights. In the spirit of open science, our implementations and dataset are released[1], hoping to facilitate future research.

2 PRELIMINARY

Problem Definition. Presented with a smart contract function f in bytecode, our goal is to learn a student network $\mathbb{S}$ that takes only bytecode as input and is able to predict its label $\mathcal{Y} \in \{0, 1\}$. $\mathcal{Y} = 1$ denotes f has a certain type of vulnerability and $\mathcal{Y} = 0$ indicates f is safe. We are interested to know whether $\mathbb{S}$ could achieve more precise predictions after the collaborative training with a teacher network $\mathbb{T}$ fed with both source code and bytecode. In this work, we concentrate on the following four types of common vulnerabilities.

Reentrancy is a well-known vulnerability that causes the notorious DAO attack [12]. When a function f_1 transfers money to a recipient contract C, due to the default settings of smart contracts, the fallback function f_b of C will be automatically triggered [24]. f_b may invoke f_1 to conduct an illegal second-time transfer. As the current execution of f_1 waits for the first-time transfer to finish, the balance of C may not be reduced yet, making f_1 wrongly believe that C still has enough balance and transfer to C again. More specifically, the expected execution trace is $f_1 \xrightarrow{transfer} C \xrightarrow{trigger} f_b \rightarrow end$, whereas the actual trace is $(f_1 \xrightarrow{transfer} C \xrightarrow{trigger} f_b)^{(N)} \rightarrow end$. As a result, attackers can exploit the reentrancy vulnerability to succeed in stealing extra Ether for $N - 1$ times.

Timestamp dependence vulnerability exists when a function uses block timestamps as a condition to perform critical operations, *e.g.,* using *block.timestamp* of a future *block* to determine the winner of a gambling game. The miner who mines the block has the capacity to alter the timestamp of the *block* within a short time interval (roughly 900 seconds) [50]. Therefore, malicious miners may manipulate the block timestamp to gain illegal profits.

Integer overflow/underflow happens when an arithmetic operation attempts to generate a numeric value that is outside the range of the *integer* type. For example, if a number v is of type *uint8*, its value is stored as a 8-bits unsigned number ranging from 0 to $2^8 - 1$. When we try to assign a value out of this range to v, that is, either larger than 255 or lower than 0, the integer overflow/underflow vulnerability will occur.

Delegatecall is almost identical to a classical function *call* method but with a critical difference. It endows the *caller* with the ability to put the code of the *callee* contract into the current execution environment of the *caller* contract [40]. However, the

[1] Code is available at https://github.com/Messi-Q/WWW2023

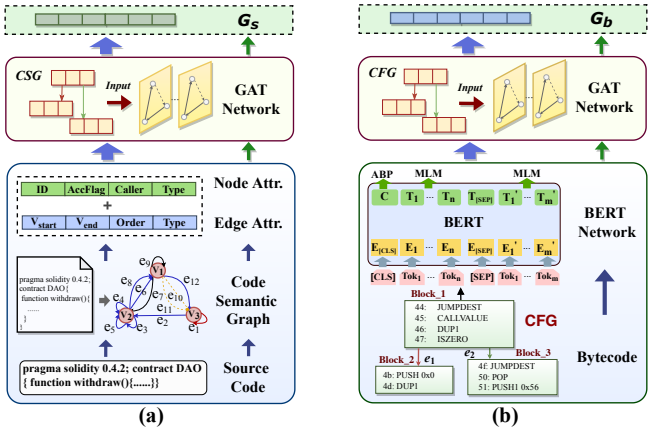


Figure 1: Illustration of extracting the graph semantic embeddings of source code and bytecode, *i.e.,* $\mathcal{G}_s$ and $\mathcal{G}_b$, respectively. (a) A semantic graph extractor and a GAT network are designed to handle the source code. (b) The BERT model and a GAT network are exploited to deal with the bytecode.

execution environments of the *caller* and the *callee* might be quite different from each other, running a function of the *callee* in the environment of the *caller* may lead to unexpected results. Attackers may exploit the characteristic of *delegatecall* to perform malicious activities. As such, we need to evaluate if a *delegatecall* will indeed cause losses, *e.g.,* Ether frozen [3].

Why focus on these vulnerabilities. We mainly focus on the 4 aforementioned vulnerabilities since: **(i)** In real attacks, 70% of financial losses in Ethereum smart contracts are caused by these vulnerabilities [5]. **(ii)** Existing works [15, 31, 33] have shown that these vulnerabilities are more common in Ethereum smart contracts. **(iii)** They manifest the typical characteristics of Ethereum smart contract vulnerabilities, *e.g.,* lack of run-time checks (*integer overflow/underflow*), lack of privilege controls (*delegatecall*), misusage of on-chain information (*timestamp dependence*), and lack of care for interactions across different contracts (*reentrancy*).

3 OUR APPROACH

Method Overview. The overall pipeline of our proposed framework consists of three key components: 1) a code semantic-modeling module, 2) a teacher-student framework, and 3) the cross-modality mutual learning strategy. Specifically, in the *training* stage, we are given a set of labeled smart contract functions with both bytecode and source code available. We first develop automated tools to cast the source code and the bytecode of a smart contract into a code semantic graph and a control flow graph, respectively. Their graph features are extracted by using graph attention networks. Thereafter, we construct a teacher-student framework, which contains a dual-modality teacher network $\mathbb{T}$ and a single-modality student network $\mathbb{S}$. Finally, we leverage a mutual learning loss and two transfer losses to collaboratively train the two networks. In the *inference* stage, when presented with bytecode merely, $\mathbb{S}$ is able to predict the missing source code embeddings and combine both modalities to approach more accurate detection. We would like to highlight that the student network $\mathbb{S}$ is the main network and the teacher network $\mathbb{T}$ is only used in the training stage. In what follows, we will elaborate on the three components, respectively.

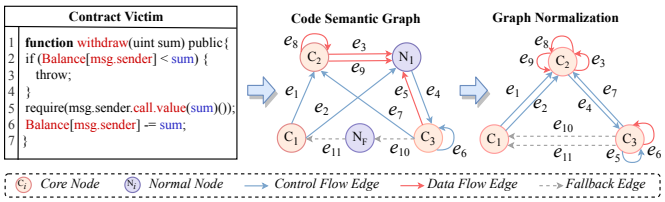


Figure 2: The first figure shows the source code of a smart contract function, while the second and third figures illustrate the process of code semantic graph construction and normalization, respectively.

3.1 Code Semantic-Modeling Module

Existing works [2, 52] have shown that programs can be characterized as symbolic graphs, which are able to preserve rich structural and semantic information. Inspired by this, we transform the source code and the bytecode of a smart contract into specific graphs, and then adopt graph attention networks to handle them for extracting the graph features. The overview of graph embedding extraction is illustrated in Figure 1.

3.1.1 Graph Processing. For *source code*, we design a code semantic graph (CSG) to frame the control and data dependencies in the contract code. Following [24], we extract two kinds of graph nodes and three types of edges. Nodes in CSG symbolize different program elements such as key function calls and variables, while edges capture the control and data flow connections between nodes. Particularly, each edge has a temporal order, which is consistent with its sequential order in the code.

For *bytecode*, we extract the control flow graph (CFG), which comprises bytecode *blocks* (*i.e.*, nodes) and control flow edges. A bytecode *block* contains a set of instructions. Prior work [47] has revealed the success of the BERT model in handling program instructions. Heuristically, we resort to the BERT network for encoding the bytecode *block*. **(1)** We pre-train a BERT model through an instruction-level task and a block-level task. (i) For instructions inside a block, a masked language task (MLM) is engaged to obtain the instruction-level information. (ii) For blocks that are connected to neighbors, we adopt an adjacency block prediction task (ABP) to capture the control flow information and relationships between adjacent blocks. **(2)** Considering the discrepancy between different vulnerabilities, we enforce a dedicated fine-tuning task of the pre-trained BERT on each type of vulnerability. **(3)** Finally, the features of bytecode blocks are extracted by the fine-tuned BERT. Notably, we have put the training details of BERT in Appendix C.

Code Semantic Graph Construction. To clearly illustrate how to characterize the source code of a smart contract function into a code semantic graph, we provide a simplified example in Figure 2. Taking contract *Victim* as an example, suppose we are to evaluate whether its *withdraw* function possesses a reentrancy vulnerability. As shown in the middle of Figure 2, function *withdraw* is first modeled as a core node C_1 since its inner code contains the *call.value* invocation (which is denoted as a key call). Then, following the temporal order of the code, we treat the critical state variable *Balance*[*msg.sender*] as a core node C_2, while the local variable *sum* is modeled as a normal node N_1. The invocation to

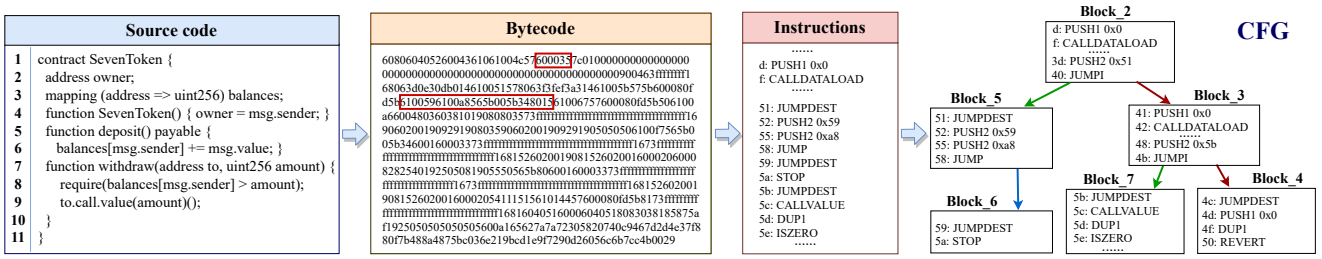


Figure 3: The first figure shows the contract source code, while its compiled bytecode is depicted in the second figure. The third figure shows the corresponding bytecode instructions. The fourth figure illustrates the construction of the control flow graph.

call.value is also extracted as a core node C_3, and the fallback function of a virtual attack contract is characterized by the normal node N_f. To capture rich semantic dependencies between these nodes, we construct three categories of edges, namely *control flow*, *data flow*, and *fallback* edges [25]. Each edge describes a path that might be traversed through by the function under test, and the temporal number of edges stands for its sequential order in the function.

Considering different functions yield graphs with distinct structures, we are motivated by [52] and normalize the code semantic graph by removing all normal nodes and merging their features into their nearest core nodes. For example, normal node N_1 in the second figure of Figure 2 is removed, with its feature aggregated to the nearest core nodes C_2 and C_3. For a normal node that has multiple nearest core nodes, its feature is passed to all of them. The edges connected to the removed normal nodes are preserved but with their start or end nodes moving to the corresponding core nodes. The third figure of Figure 2 illustrates the normalized graph of the second figure of Figure 2. As graph neural networks are usually flat in propagating information, the normalization process also helps highlight the core nodes.

Control Flow Graph Construction. To clearly depict the construction process of the bytecode control flow graph, we present a specific example in Figure 3. Given the smart contract source code, we employ a public compiler to translate it into the bytecode and develop the automated tool *BinaryCFGExtractor*[2] to extract a control flow graph of the compiled bytecode. A bytecode control flow graph (CFG) consists of bytecode basic blocks (*i.e.*, nodes) and control flow edges. **(1)** Our first insight is that the basic block comprises the sequence of EVM instructions. One basic block is connected to subsequent basic blocks through a branch instruction in the CFG [34]. In our analysis, we treat the branch instructions (*e.g.*, JUMP, JUMPI, RETURN) as the sign of the end of a basic block, which means that the branch instruction is regarded as a flag to segment the basic blocks. To further show the bytecode instructions, we list the distinctive bytecode value and its corresponding definitions and instructions in Appendix A. **(2)** Our second insight is that the basic blocks are closely related to each other by the control flow edges rather than being isolated. A control flow edge captures the control flow dependencies of a *conditional* statement or a *call* statement. **(3)** There are three main categories of control flow edges in a CFG, which are denoted as different colors in the right of Figure 3. The unconditional jump instructions (*i.e.*, JUMP) are highlighted with blue arrows. True or false conditional jump instructions (*i.e*, JUMPI) are demonstrated with green and red arrows, respectively.

[2]Code is available at https://github.com/Messi-Q/BinaryCFGExtractor

3.1.2 Graph Embedding Distillation. After obtaining the two kinds of graphs, we build upon the architecture of the graph attention network (GAT) [41] to learn the high-level graph semantic embeddings of both the source code and the bytecode, *i.e.*, $\mathcal{G}_s$ and $\mathcal{G}_b \in \mathbb{R}^d$, respectively. The graph embedding extraction consists of two phases, namely a *message propagation* phase and an *aggregation* phase. In the message propagation phase, the network passes information along edges by following their sequential orders in the code. As an example, at time step k, the message flows through the k-th temporal edge e_k and updates the hidden state of the end node of e_k. Thereafter, GAT computes the hidden states of every node by attending to its neighbors as:

$$\vec{h}_i' = \sigma\Big(\sum\nolimits_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} \vec{h}_j\Big) \tag{1}$$

where σ is a nonlinear activation function, $\mathcal{N}_i$ denotes the *neighbors* of node i in the graph, $\mathbf{W}$ is a weight matrix. α_{ij} represents the *attention coefficient* that is given by:

$$\alpha_{ij} = \frac{exp(\mathcal{T}(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \oplus \mathbf{W}\vec{h}_j]))}{\sum_{k \in \mathcal{N}_i} exp(\mathcal{T}(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \oplus \mathbf{W}\vec{h}_k]))} \tag{2}$$

where $\oplus$ denotes concatenation, $\vec{\mathbf{a}}$ is the weight vector of a single-layer MLP, and $\mathcal{T}$ is the *LeakyReLU* function. After successively traversing all edges, GAT generates the final high-level graph semantic embedding $\mathcal{G} \in \mathbb{R}^d$ by aggregating the hidden states of all participating nodes in the graph:

$$\mathcal{G} = \sum\nolimits_{i=1}^{V} \sigma(\mathbf{P}_{gate}(\mathbf{M_1}\vec{h}_i' + \mathbf{b_1})) \odot \mathbf{P}(\mathbf{M_2}\vec{h}_i' + \mathbf{b_2}) \tag{3}$$

where $\odot$ denotes the element-wise product, σ is an activation function. Matrix $\mathbf{M_j}$ and bias vector $\mathbf{b_j}$, with subscript $j \in \{1, 2\}$, are trainable network parameters. V represents the number of nodes and $\mathbf{P}$ is a multi-layer perceptron.

3.2 Teacher-Student Framework

After extracting the two kinds of graph embeddings, we introduce a novel teacher-student framework that addresses the task of enhancing smart contract vulnerability detection on bytecode in a mutual learning fashion.

Dual-Modality Teacher Network (DMT). The dual-modality teacher network $\mathbb{T}$ takes the two graph embeddings, $\mathcal{G}_s$ and $\mathcal{G}_b$, as inputs. As depicted in the right of Figure 4, **(1)** the teacher network builds a *semantic extractor*, which utilizes a 3-layer CNN, to process the graph embeddings. The convolution filter size is set to 1×3 and the numbers of filters are set to 64, 128, and 256, respectively. **(2)** Batch normalization (BN), rectified linear unit (ReLU), and max pooling are employed after each CNN layer,

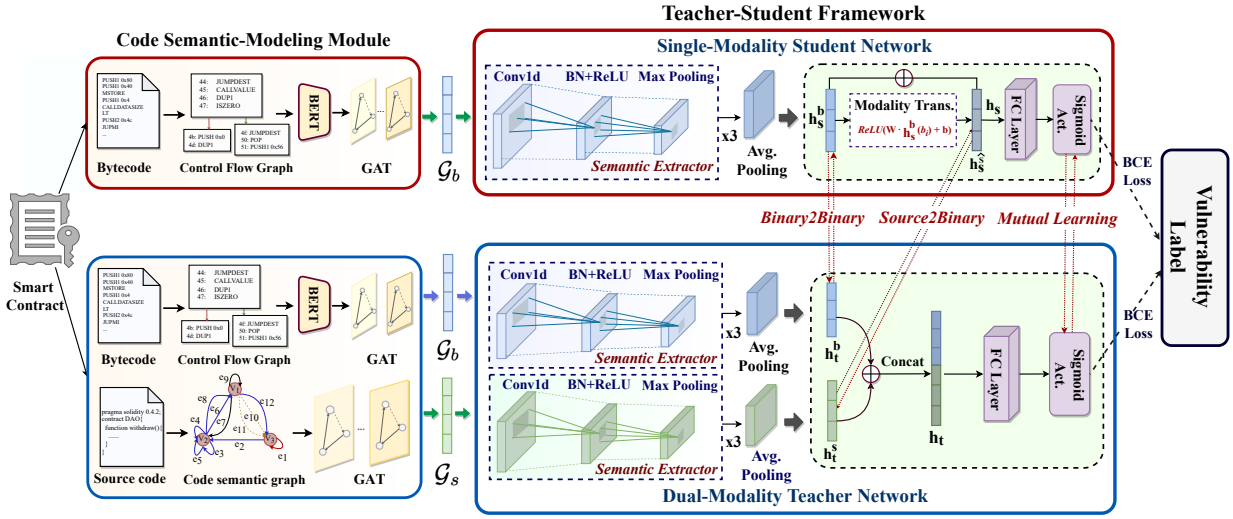


Figure 4: A high-level overview of our pipeline. (1) Code semantic-modeling module, which casts the source code and the bytecode into a code semantic graph and a control flow graph, respectively, and adopts a graph attention network (GAT) to extract graph embeddings. (2) Teacher-student framework, which consists of a dual-modality teacher network and a single-modality student network. A mutual learning loss and two transfer losses are engaged to collaboratively train the two networks.

which highlights the significant elements and avoids overfitting. **(3)** The two graph embeddings are then passed into a global average pooling layer respectively to generate the semantic intermediate representations of the source code and the bytecode, $\mathbf{h}_\mathbf{t}^\mathbf{s}$ and $\mathbf{h}_\mathbf{t}^\mathbf{b}$. Next, $\mathbf{h}_\mathbf{t}^\mathbf{s}$ and $\mathbf{h}_\mathbf{t}^\mathbf{b}$ are fused by concatenation, *namely* $\mathbf{h_t} = \mathbf{h}_\mathbf{t}^\mathbf{s} \oplus \mathbf{h}_\mathbf{t}^\mathbf{b}$. **(4)** The fused feature vector $\mathbf{h_t}$ is finally fed into a fully connected layer with a *sigmoid* activation to output the label $\mathcal{Y}_t$.

Single-Modality Student Network (SMS). The single-modality student network $\mathbb{S}$ takes the graph embedding $\mathcal{G}_b$ of the bytecode as input. Technically, we adopt the sub-network of the teacher as the student network, but with particular modifications to support the cross-modality knowledge transfer. We use the intermediate representations learned by the teacher network to supervise the learning of the student network. Let $\mathbf{h}_\mathbf{s}^\mathbf{b}$ denote the semantic intermediate representations of the bytecode generated by the student network, we model a transfer loss *B2B* within the bytecode modality as:

$$\mathcal{L}_{B2B} = \sum_{i=1}^{N} \left\| \mathbf{h}_\mathbf{t}^\mathbf{b}(b_i) - \mathbf{h}_\mathbf{s}^\mathbf{b}(b_i) \right\|^2 \tag{4}$$

where N is the number of functions. Since the *B2B* loss can propagate knowledge only in the bytecode modality, we proceed to explore learning cross-modality correlations between source code and bytecode. Specifically, we leverage global average pooling to summarize the representations of input graph embeddings. Then, the global context of the source code $\mathbf{h}_\mathbf{t}^\mathbf{s}$ in $\mathbb{T}$ and the bytecode $\mathbf{h}_\mathbf{s}^\mathbf{b}$ in $\mathbb{S}$ can be acquired. The main idea is to constrain the global context representations of the paired modalities (*viz.*, source code and bytecode) to be similar to each other. Explicitly, we design a dedicated *modality transformer* layer to endow $\mathbb{S}$ with the ability to reconstruct semantic intermediate representations of the source code from bytecode features as $\mathbf{h}_\mathbf{s}^{\hat{\mathbf{s}}}(b_i) = ReLU(\mathbf{W_s} \cdot \mathbf{h}_\mathbf{s}^\mathbf{b}(b_i) + \mathbf{b_s})$, where $\mathbf{W_s}$ and $\mathbf{b_s}$ denote the weight matrix and the bias vector, respectively. Finally, we model a transfer loss *S2B* to cross the two modalities between $\mathbb{T}$ and $\mathbb{S}$ as:

$$\mathcal{L}_{S2B} = \sum_{i=1}^{N} \left\| \mathbf{h}_\mathbf{t}^\mathbf{s}(s_i) - \mathbf{h}_\mathbf{s}^{\hat{\mathbf{s}}}(b_i) \right\|^2 \tag{5}$$

Similar to $\mathbb{T}$, we fuse $\mathbf{h}_\mathbf{s}^\mathbf{b}(b_i)$ and $\mathbf{h}_\mathbf{s}^{\hat{\mathbf{s}}}(b_i)$ in $\mathbb{S}$, and pass the concatenated feature $\mathbf{h_s}$ into a fully connected layer with a *sigmoid* activation to output the label $\mathcal{Y}_s$.

3.3 Cross-Modality Mutual Learning Strategy

To achieve effective knowledge transfer from $\mathbb{T}$ to $\mathbb{S}$, inspired by [46], we present a cross-modality mutual learning strategy to collaboratively train them. Particularly, we compute the mutual learning losses by using the binary cross-entropy (BCE) loss between the labels of the teacher network ($\mathcal{Y}_t$), the student network ($\mathcal{Y}_s$), and the ground-truth ($\mathcal{Y}$):

$$\mathcal{L}_{mutual}^{teacher} = BCE(\mathcal{Y}, \mathcal{Y}_t) + BCE(\mathcal{Y}_t, \mathcal{Y}_s) \tag{6}$$

$$\mathcal{L}_{mutual}^{student} = BCE(\mathcal{Y}, \mathcal{Y}_s) + BCE(\mathcal{Y}_s, \mathcal{Y}_t) \tag{7}$$

where $L_{mutual}^{teacher}$ and $L_{mutual}^{student}$ denote the supervised losses of $\mathbb{T}$ and $\mathbb{S}$, respectively. Finally, by combining the mutual learning losses with the two transfer losses *B2B* and *S2B*, we obtain the overall losses of the two networks by:

$$\mathcal{L}_{teacher} = \lambda_t \mathcal{L}_{mutual}^{teacher} + \gamma_t \mathcal{L}_{B2B} + \omega_t \mathcal{L}_{S2B} \tag{8}$$

$$\mathcal{L}_{student} = \lambda_s \mathcal{L}_{mutual}^{student} + \gamma_s \mathcal{L}_{B2B} + \omega_s \mathcal{L}_{S2B} \tag{9}$$

where λ, γ, and ω are tunable network parameters for balancing different losses. An important highlight in our framework is that not only $\mathbb{S}$ learns from its teacher, but also $\mathbb{T}$ can benefit from the student via cross-modality mutual learning. Experiments in §4.4 show that such a cross-modality mutual learning strategy could contribute to promising performance gains. This may stem from the fact that cross-modality mutual learning helps align the global contexts in different modalities better in the feature space compared to conventional one-way training. The teacher network and student

Table 1: Performance comparison (%) in terms of *accuracy* (ACC), *recall* (RE), *precision* (PRE), and *F1-score* (F1). Fourteen methods are included in the comparisons. 'n/a' means the corresponding tool does not support detecting the vulnerability type.

Methods	Reentrancy				Timestamp				Overflow/Underflow				Delegatecall			
	ACC	RE	PRE	F1	ACC	RE	PRE	F1	ACC	RE	PRE	F1	ACC	RE	PRE	F1
sFuzz [30]	55.69	14.95	10.88	12.59	33.41	27.01	23.15	24.93	45.50	25.97	25.88	25.92	64.37	47.22	58.62	52.31
Smartcheck [37]	54.65	16.34	45.71	24.07	47.73	79.34	47.89	59.73	53.91	68.54	42.81	52.70	62.41	56.21	45.56	50.33
Osiris [38]	56.73	63.88	40.94	49.90	66.83	55.42	59.26	57.28	68.41	34.18	60.83	43.77	n/a	n/a	n/a	n/a
Oyente [27]	65.07	63.02	46.56	53.55	68.29	57.97	61.04	59.47	69.71	57.55	58.05	57.80	n/a	n/a	n/a	n/a
Mythril [29]	64.27	75.51	42.86	54.68	62.40	49.80	57.50	53.37	n/a	n/a	n/a	n/a	75.06	62.07	72.30	66.80
Securify [40]	72.89	73.06	68.40	70.41	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Slither [14]	74.02	73.50	74.44	73.97	68.52	67.17	69.27	68.20	n/a	n/a	n/a	n/a	68.97	52.27	70.12	59.89
Vanilla-RNN [35]	65.90	72.89	67.39	70.03	64.41	65.17	64.16	64.66	68.12	70.19	67.00	68.56	64.33	67.26	63.77	65.47
ReChecker [32]	70.95	72.92	70.15	71.51	66.65	54.53	73.37	62.56	70.49	71.59	70.56	71.07	67.98	70.66	66.47	68.50
GCN [20]	73.21	73.18	74.47	73.82	75.91	77.55	74.93	76.22	67.53	70.93	69.52	70.22	65.76	69.74	69.01	69.37
TMP [52]	76.45	75.30	76.04	75.67	78.84	76.09	78.68	77.36	70.85	69.47	70.26	69.86	69.11	70.37	68.18	69.26
AME [24]	81.06	78.45	79.62	79.03	82.25	80.26	81.42	80.84	73.24	71.59	71.36	71.47	72.85	69.40	70.25	69.82
SMS	83.85	77.48	79.46	78.46	89.77	91.09	89.15	90.11	79.36	72.98	78.14	75.47	78.82	73.69	76.97	75.29
DMT	89.42	81.06	83.62	82.32	94.58	96.39	93.60	94.97	85.64	74.32	85.44	79.49	82.76	77.93	84.61	81.13

network in the process of mutual learning are optimized jointly in every mini-batch, obtaining better performance in bug detection.

4 EVALUATION

In this section, we present extensive evaluations on our proposed framework. We seek to address the following research questions.

- **RQ1**: Can our proposed method effectively detect the four types of smart contract vulnerabilities? How is its performance compared with state-of-the-art approaches?
- **RQ2**: Is the cross-modality mutual learning strategy helpful to improve the performance of vulnerability detection on bytecode?
- **RQ3**: How do our designed code semantic-modeling module and teacher-student network contribute to the whole framework?

In the following, we first introduce the experimental setup, followed by answering the above research questions one by one.

4.1 Experimental Setup

Datasets. We notice that there is still a lack of datasets for smart contract vulnerability detection. Indeed, it is *labor-intensive* and *time-consuming* to collect and label a large-scale smart contract dataset. Most existing works either publish unlabeled datasets or a small number of labeled contracts, which is insufficient for model training. Towards this, we construct and release a benchmark dataset that concerns four types of vulnerabilities, namely *reentrancy, timestamp dependence, integer overflow/underflow, and delegatecall*. This dataset was created by collecting smart contracts from three different sources, i.e., Ethereum platform (over 96%), GitHub repositories, and blog posts that analyze contracts. We collected 514,880 functions (from 42,910 smart contracts) that have both source code and bytecode available. These functions are labeled manually. Detailed labeling strategies are introduced in Appendix B. In the experiments, we select 80% of functions as the training set and the rest 20% as the test set. We repeat each experiment five times and report average results.

Implementations. Our system consists of three components: 1) the automated tools, *SourceCSGExtractor* and *BinaryCFGExtractor*, for extracting graphs of smart contracts, 2) the BERT model and the graph attention network for extracting graph embeddings, and 3) the teacher-student framework. Specifically, the *SourceCSGExtractor* and *BinaryCFGExtractor* are implemented with Python, where *SourceCSGExtractor* realizes a semantic graph extractor from source code while *BinaryCFGExtractor* integrates the bytecode CFG analyzer and the symbolic execution solver of an off-the-shelf tool termed *Octopus* [1]. We accomplish the BERT model and the GAT network with PyTorch, where their hidden layer sizes are set to 256. The teacher-student framework is implemented with PyTorch. The two networks are composed of a 3-layer CNN followed by batch normalization, activation, and max-pooling layers.

Parameter Settings. All experiments are conducted on a computer equipped with an Intel Core i9 CPU at 3.3GHz, a GPU at 2080Ti, and 64GB Memory. Adam optimizer is employed in the proposed networks. We apply a grid search to find the best hyper-parameters: the learning rate l is tuned amongst {0.0001, 0.0005, 0.001, 0.002}, the hidden layer size h is searched in {64, 128, 256, 512}, and batch size β in {16, 32, 64, 128}. We choose a set of hyper-parameters that achieve the best performance on the training set. We report the performance with the default settings: 1) $l = 0.001$, 2) $h = 256$, and 3) $\beta = 64$. The balancing factors λ, γ, and ω which are tuned to weigh different functions have been empirically set to 1.0 throughout the experiments.

4.2 Performance Comparison (RQ1)

Comparison with Conventional Bug Detection Tools. We first benchmark the proposed approach against the conventional vulnerability detection tools, including *sFuzz* [30], *Smartcheck* [37], *Osiris* [38], *Oyente* [27], *Mythril* [29], *Securify* [40], and *Slither* [14]. Here, we do not compare with several related works, which are either 1) not open-sourced [44], or 2) handling other types of smart contracts like EOSIO [16], or 3) focusing on different vulnerability types than ours [8, 39]. Quantitative experimental results are summarized in Table 1. From the table, we obtain the following observations. *First*, the conventional detection tools have not yet achieved high accuracy on the four vulnerabilities. For example, for reentrancy vulnerability, *sFuzz* and *Smartcheck* only achieve 55.69% and 54.65% accuracy, while *Securify* and *Slither* obtain 72.89% and 74.02% accuracy. This may stem from two facts: 1) vulnerability detection from bytecode is inherently challenging since the bytecode conveys only instruction-level binary code, and 2) current tools (such as *sFuzz* and *Osiris*) concentrate on utilizing only low-level instruction information and are unable to incorporate high-level semantics from the source code.

Table 2: Performance comparison (%) of the SMS trained using different strategies.

Training Strategy	Reentrancy			Timestamp			Overflow/Underflow			Delegatecall		
	ACC	F1	ACC Decrease	ACC	F1	ACC Decrease	ACC	F1	ACC Decrease	ACC	F1	ACC Decrease
SMS	83.85	78.46	—	89.77	90.11	—	79.36	75.47	—	78.82	75.29	—
three losses - w/o	78.25	71.38	-5.60	83.82	83.46	-5.95	74.06	70.18	-5.30	72.90	69.24	-5.92
B2B - w/o	81.43	74.54	-2.42	88.39	88.29	-1.38	77.46	73.02	-1.90	76.57	73.68	-2.25
S2B - w/o	81.04	75.86	-2.81	88.32	88.43	-1.45	77.03	73.14	-2.33	77.18	74.02	-1.64
mutual learning - w/o	82.05	76.58	-1.80	88.25	88.36	-1.52	77.51	73.53	-1.85	76.72	73.65	-2.10

Table 3: Performance comparison (%) between SMS and its variants on the four vulnerabilities.

Variants	Reentrancy				Timestamp				Overflow/Underflow				Delegatecall			
	ACC	RE	PRE	F1	ACC	RE	PRE	F1	ACC	RE	PRE	F1	ACC	RE	PRE	F1
SMS(GPT)	74.32	72.47	75.50	73.95	80.64	80.97	82.83	81.89	75.07	70.38	75.96	73.06	74.76	71.89	72.52	72.20
SMS(Word2vec)	84.08	75.49	81.58	78.42	85.12	87.82	82.12	84.87	78.12	71.90	76.18	73.98	77.26	72.41	74.38	73.38
SMS(GCN)	73.85	72.82	71.84	72.33	78.65	79.84	77.19	78.49	74.64	69.83	70.27	70.05	72.62	70.69	73.16	71.90
SMS(1-CNN)	78.87	72.95	75.56	74.23	81.08	81.57	80.77	81.17	73.01	67.67	69.87	68.75	72.06	70.34	70.00	70.17
SMS(2-CNN)	81.86	76.94	78.94	77.93	81.24	83.98	79.92	81.90	76.45	69.10	70.33	69.71	76.67	72.11	75.99	74.00
SMS(FC)	78.94	76.24	76.81	76.52	85.22	89.39	87.18	82.35	77.75	70.80	72.22	71.50	68.26	68.47	71.81	70.10
SMS(RNN)	80.63	76.37	78.96	77.64	82.24	83.84	82.85	83.34	77.45	69.44	71.56	70.48	76.34	72.66	73.38	73.02
SMS(Tanh)	83.58	77.74	78.06	77.90	89.29	87.74	87.15	87.44	78.50	73.21	77.93	75.50	74.77	68.79	73.57	71.10
SMS	83.85	77.48	79.46	78.46	89.77	91.09	89.15	90.11	79.36	72.98	78.14	75.47	78.82	73.69	76.97	75.29

Next, we evaluate the performance gain of the proposed methods against state-of-the-art tools. Surprisingly, we found that the results of the single-modality student network (SMS), are quite encouraging. More specifically, it keeps delivering the best performance in all the four metrics on each type of vulnerability, and the relative accuracy gains on *reentrancy*, *timestamp dependence*, *integer overflow/underflow*, and *delegatecall* over the state-of-the-art tool are 2.79%, 7.52%, 6.12%, and 5.97%, respectively.

Comparison with Deep Learning-Based Methods. We further compare our method to other deep learning alternatives, *namely* Vanilla-RNN [35], ReChecker [32], GCN [20], TMP [52], and AME [24]. For a feasible comparison, Vanilla-RNN and ReChecker are fed with the bytecode sequences, while GCN, TMP, and AME are presented with the bytecode CFG. We illustrate the performance of deep learning models in the middle of Table 1. Quantitative results reveal that Vanilla-RNN and ReChecker have relatively poor performance. Graph neural networks, GCN, TMP, and AME, which can capture graph structural information, deliver better performance. Technically, we speculate that the deep learning-based methods still have difficulties in coping with pure bytecode. In terms of *F1-score*, SMS consistently outperforms other methods by a large margin on the four types of vulnerabilities. Empirical evidences clearly reveal the potential of using a teacher network to supervise the learning of a student network, which leads to impressive performance gains. Moreover, the high accuracy obtained by the dual-modality teacher (DMT) network suggests that it is useful to combine both information of the source code and the bytecode.

4.3 Evaluation on Mutual Learning (RQ2)

By default, our cross-modality mutual learning losses consist of three loss functions. To evaluate the effectiveness of the mutual learning strategy, we modify the models by removing one of the losses at each time (i.e., *mutual learning* loss, *B2B* loss, and *S2B* loss) and report the results in Table 2.

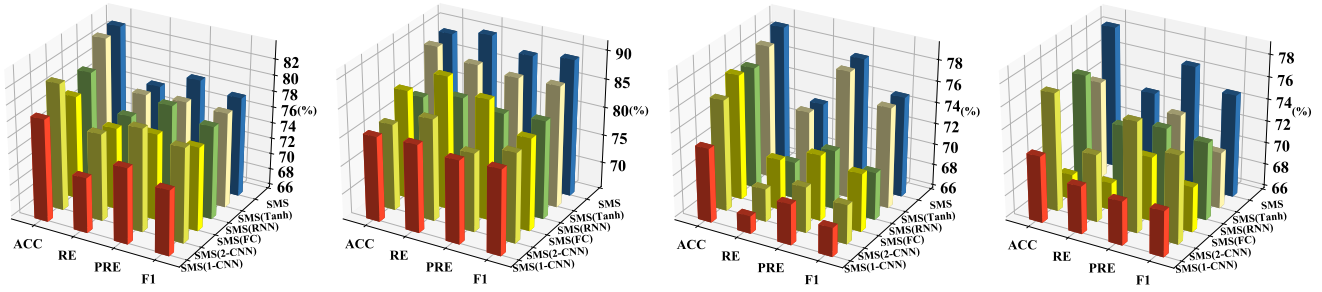
Notably, without the *mutual learning* loss, the accuracy obtained by SMS decreases by 1.80%, 1.52%, 1.85%, and 2.10% on the four types of vulnerabilities, respectively. This indicates that the student network indeed benefits from the teacher network and gains performance improvements through cross-modality mutual learning losses. Furthermore, without the transfer losses of *B2B* and *S2B* in the overall losses, the accuracy acquired by the student network decreases by as much as 2.81%, which shows the efficacy of the transfer losses. Technically, the *B2B* loss and the *S2B* loss address the modality inconsistency between source code and bytecode by aligning the two modalities in the high-level feature space. In addition, we evaluate the effect of removing all the three losses and observe that the performance degenerates significantly. In summary, our teacher-student network achieves effective performance gains via the cross-modality mutual learning strategy.

4.4 Ablation Study (RQ3)

Study on Code Semantic-Modeling Module. Our code semantic-modeling module utilizes a pre-trained BERT model for feature preprocessing and a graph attention network for graph embedding extraction. To evaluate the two components, we empirically investigate several variants and conduct comparing experiments, with results listed in Table 3.

For *feature preprocessing*, prior works [11] have confirmed that features processed by a pre-trained model (*e.g.,* Word2vec, GPT, BERT) can achieve better performance than hand-crafted features. This demonstrates that building a suitable model for feature preprocessing is feasible yet effective. Therefore, we use the BERT model to preprocess the bytecode features in the code semantic-modeling module. Compared with Word2vec and GPT, BERT with a *masked language* task and an *adjacency block prediction* task considers not only instruction-level but also block-level information of the bytecode CFG [48]. Meanwhile, the bidirectional transformer in BERT embraces the ability to extract bidirectional information. For *graph embedding extraction*, we resort to the graph attention network in the code semantic-modeling module. It is worth mentioning that traditional graph neural networks such as GCN fail to emphasize the distinct importance of different nodes, which is explicitly considered in GAT by using an attention mechanism. Empirical results suggest that GAT contributes to better performance in extracting the graph embeddings.

Study on Teacher-Student Framework. By default, our proposed teacher-student framework adopts a network structure with a 3-layer CNN. Each CNN layer is followed by batch normalization,



(a) Reentrancy (b) Timestamp (c) Overflow/Underflow (d) Delegatecall

Figure 5: Visually comparison of SMS and its variants on the four types of vulnerabilities.

rectified linear unit, and max-pooling layers. To validate such kinds of network architectures, we further try five other alternatives. *First*, we keep the batch normalization, rectified linear unit, and max-pooling layers, but change the 3-layer CNN to 1 or 2 layers. The two variants are denoted as SMS(1-CNN) and SMS(2-CNN), respectively. *Then*, we replace the convolution layer with a fully connected layer, which we denote as SMS(FC). We also try replacing them with a RNN layer, which we term as SMS(RNN). *Finally*, we replace the ReLU activation layer with Tanh activation layer while keeping the other layers fixed. This variant is denoted as SMS(Tanh). We list the quantitative results in the middle of Table 3 and further visualize the results in Figure 5. We may observe that: 1) the default settings of SMS yield better results, 2) using a Tanh activation layer or changing the number of CNN layers leads to a slight performance drop, and 3) adopting RNN structure in the teacher network and student network does not translate to performance gain.

5 RELATED WORK

Smart Contract Vulnerability Detection. Traditional efforts for smart contract vulnerability detection mostly revolve around static analysis and dynamic analysis methods. **(1)** Static analysis can be further divided into formal verification, program analysis, and symbolic execution. For example, [4] proposes a formal model to verify smart contract bytecode by using the Isabelle/HOL tool. [27] performs symbolic execution on smart contracts and checks bugs based on expert-defined rules. [40] conducts advanced program analysis to infer semantic facts of data-flows in smart contracts. [15] develops an overflow detector that adopts a taint analysis-based tracking technique to detect potential overflow vulnerabilities in Ethereum. **(2)** Dynamic analysis methods discover potential vulnerabilities in smart contracts by executing the contract code. [33] introduce *Serum*, which exploits dynamic taint tracking to monitor data-flows during contract execution to automatically detect and prevent basic and advanced reentrancy attacks. [30] presents *sFuzz*, which identifies vulnerabilities by adopting a branch distance-driven fuzzing technique. *Smartian* [9] leverages the data-flow-based feedback to find meaningful transaction sequence orders, triggering more contract states to detect vulnerabilities. [26] introduces a fully automatic fuzzing framework equipped with invocation ordering and crucial branch revisiting to detect smart contract bugs.

Recent attempts have explored using deep learning networks for vulnerability detection. [35] employs a sequential model to handle smart contract bytecode sequences. [13] uses a dynamic vulnerability detection framework that extracts features from transaction data and classifies harmful transactions using machine learning models. [52] proposes to cast the contract source code into a graph structure and construct graph neural networks as the detection model. [24] explores combining deep learning with expert patterns to detect vulnerabilities in an explainable fashion. [49] presents a hybrid deep learning model, which combines features from different models to detect smart contract bugs. In this work, we go further to explore a novel deep learning-based scheme, *i.e., cross-modal mutual learning*, to improve smart contract bug detection.

Teacher-Student Network. Existing research [42] has shown that transferring knowledge from a teacher network to a student network is beneficial to improve the performance of the student network. A teacher network often refers to a heavy, cumbersome model, while a student network refers to a simple, lightweight model. [21] designs a student network to address the math word matching task under the supervision of a teacher network. Recently, mutual learning techniques have been proposed for knowledge transfer between networks. For example, [51] proposes a binocular mutual learning framework, which achieves the compatibility of the global view and the local view. [46] presents the multi- to single-modality teacher-student network on the audio tagging task.

6 CONCLUSION

In this paper, we investigate whether mutual learning could help in the challenging scenario where smart contract source code is missing. We propose a cross-modality mutual learning strategy to collaboratively train a dual-modality teacher network and a single-modality student network. The student network, which serves as the primary network, distills knowledge from the teacher and is enhanced to reconstruct the missing modality. Extensive experiments on four different types of vulnerabilities demonstrate that our method consistently and significantly surpasses state-of-the-art tools. It is worth pointing out that our proposed approach can be extendable to other programs as long as they have the paired modalities of source code and bytecode. We have also released our implementations and a large-scale labeled benchmark dataset, hoping to push forward the boundary of this research direction.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China under Grant 2021YFB2700500, the Key R&D Program of Zhejiang Province under Grant 2022C01086 and Grant 2023C01217, and by the Scientific Research Fund of Zhejiang Provincial Education Department under Grant Y202250832.

REFERENCES

[1] 2022. Octopus. https://github.com/FuzzingLabs/octopus.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*. Springer, 164–186.

[4] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*.

[5] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–43.

[6] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2021. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering* (2021).

[7] Sijie Chen, Hanning Mi, Jian Ping, Zheng Yan, Zeyu Shen, Xuezhi Liu, Ning Zhang, Qing Xia, and Chongqing Kang. 2022. A blockchain consensus mechanism that uses Proof of Solution to optimize energy dispatch and trading. *Nature Energy* (2022), 1–8.

[8] Weimin Chen, Xinran Li, Yuting Sui, Ningyu He, Haoyu Wang, Lei Wu, and Xiapu Luo. 202. Sadponzi: Detecting and characterizing ponzi schemes in ethereum smart contracts. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 5, 2 (202), 1–30.

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

[10] CoinMarketCap. 2022. Ethereum (ETH) price, charts, market cap, and other metrics. https://coinmarketcap.com/currencies/ethereum.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[12] Vikram Dhillon, David Metcalf, and Max Hooper. 2017. The DAO hacked. In *Blockchain Enabled Applications*. Springer, 67–78.

[13] Mojtaba Eshghie, Cyrille Artho, and Dilian Gurov. 2021. Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning. In *Evaluation and Assessment in Software Engineering*. 305–312.

[14] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[15] Jianbo Gao, Han Liu, Chao Liu, Qingshan Li, Zhi Guan, and Zhong Chen. 2019. Easyflow: Keep ethereum away from overflow. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 23–26.

[16] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. {EOSAFE}: Security Analysis of {EOSIO} Smart Contracts. In *30th USENIX Security Symposium (USENIX Security 21)*. 1271–1288.

[17] Huiwen Hu and Yuedong Xu. 2021. SCSGuard: Deep Scam Detection for Ethereum Smart Contracts. *ArXiv* abs/2105.10426 (2021).

[18] Inspex. 2021. Reentrancy Attack on Cream Finance. https://inspexco.medium.com/reentrancy-attack-on-cream-finance-incident-analysis-1c629686b6f5.

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

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

[21] Zhenwen Liang and Xiangliang Zhang. 2021. Solving Math Word Problems with Teacher Supervision. In *IJCAI*. 3522–3528.

[22] Iuon-Chang Lin and Tzu-Chun Liao. 2017. A survey of blockchain security issues and challenges. *Int. J. Netw. Secur.* 19, 5 (2017), 653–659.

[23] Shaoying Liu, Honghui Li, Zhouxian Jiang, Xiuru Li, Feng Liu, and Yan Zhong. 2021. Rigorous code review by reverse engineering. *Information and Software Technology* 133 (2021), 106503.

[24] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion. In *IJCAI*. 2751–2759.

[25] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2021. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[26] Zhenguang Liu, Peng Qian, Jiaxu Yang, Lingfeng Liu, Xiaojun Xu, Qinming He, and Xiaosong Zhang. 2023. Rethinking Smart Contract Fuzzing: Fuzzing With Invocation Ordering and Important Branch Revisiting. *IEEE Transactions on Information Forensics and Security* 18 (2023), 1237–1251.

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

[28] Giovanni Mirabelli and Vittorio Solina. 2020. Blockchain and agricultural supply chains traceability: research trends and future challenges. *Procedia Manufacturing* 42 (2020), 414–421.

[29] Bernhard Mueller. 2017. A framework for bug hunting on the Ethereum blockchain. https://github.com/ConsenSys/mythril.

[30] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

[31] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph K. Liu, and R. Doss. 2019. Security Analysis Methods on Ethereum Smart Contract Vulnerabilities: A Survey. *ArXiv* abs/1908.08605 (2019).

[32] Peng Qian, Zhenguang Liu, Qinming He, Roger Zimmermann, and Xun Wang. 2020. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access* 8 (2020), 19685–19695.

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

[34] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2021. EVMPatch: timely and automated patching of ethereum smart contracts. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*.

[35] Wesley Joon-Wie Tann, X. Han, Sourav Sengupta, and Y. Ong. 2018. Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Vulnerabilities. *ArXiv* abs/1811.06632 (2018).

[36] Parity Technologies. 2021. Security alert: Parity wallet (multi-sig wallets). https://www.parity.io/security-alert-2/.

[37] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 9–16.

[38] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 664–676.

[39] Christof Ferreira Torres, Mathis Steichen, et al. 2019. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. 1591–1607.

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

[41] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[42] Xionghui Wang, Jian-Fang Hu, Jian-Huang Lai, Jianguo Zhang, and Wei-Shi Zheng. 2019. Progressive teacher-student learning for early action prediction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3556–3565.

[43] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[44] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1398–1409.

[45] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. 2019. Blockchain technology overview. *arXiv preprint arXiv:1906.11078* (2019).

[46] Yifang Yin, Harsh Shrivastava, Ying Zhang, Zhenguang Liu, Rajiv Ratn Shah, and Roger Zimmermann. 2021. Enhanced Audio Tagging via Multi-to Single-Modal Teacher-Student Mutual Learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 10709–10717.

[47] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 1145–1152.

[48] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems* 33 (2020), 3872–3883.

[49] Lejun Zhang, Weijie Chen, Weizheng Wang, Zilong Jin, Chunhui Zhao, Zhennao Cai, and Huiling Chen. 2022. Cbgru: A detection method of smart contract vulnerability based on a hybrid model. *Sensors* 22, 9 (2022), 3577.

[50] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. {TXSPECTOR}: Uncovering attacks in ethereum from transactions. In *29th USENIX Security Symposium (USENIX Security 20)*. 2775–2792.

[51] Ziqi Zhou, Xi Qiu, Jiangtao Xie, Jianan Wu, and Chi Zhang. 2021. Binocular Mutual Learning for Improving Few-shot Classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8402–8411.

[52] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart Contract Vulnerability Detection using Graph Neural Network. In *IJCAI*. 3283–3290.

APPENDIX

A DEFINITIONS OF BYTECODE VALUES AND INSTRUCTIONS

We present the 11 categories of bytecode values and their corresponding definitions. Moreover, we list the distinctive opcodes used as features to represent the binary instruction operations.

- 0x00 - 0x0B: `STOP`, `ADD`, `MUL`, `SUB`, `DIV`, `SDIV`, `MOD`, `SMOD`, `ADDMOD`, `MULMOD`, `EXP`, `SIGNEXTEND` (Stop and Arithmetic Operations).
- 0x10 - 0x1A: `LT`, `GT`, `SLT`, `SGT`, `EQ`, `ISZERO`, `AND`, `OR`, `XOR`, `NOT`, `BYTE`, `SHL`, `SHR`, `SAR` (Comparison and Bitwise Logic Operations).
- 0x20: `KECCAK256` (KECCAK256 Method).
- 0x30 - 0x3E: `ADDRESS`, `BALANCE`, `ORIGIN`, `CALLER`, `CALLVALUE`, `CALLDATALOAD`, `CALLDATASIZE`, *etc* (Environmental Information).
- 0x40 - 0x45: `BLOCKHASH`, `COINBASE`, `TIMESTAMP`, `NUMBER`, `DIFFICULTY`, `GASLIMIT`, `CHAINID`, *etc* (Block Information).
- 0x50 - 0x5B: `POP`, `MLOAD`, `MSTORE`, `MSTORE8`, `SLOAD`, `SSTORE`, `JUMP`, `JUMPI`, `PC`, *etc* (Stack, Memory, Storage and Flow Operations).
- 0x60 - 0x7F: `PUSH1` – `PUSH32` (Push Operations).
- 0x80 - 0x8F: `DUP1` – `DUP16` (Duplication Operations).
- 0x90 - 0x9F: `SWAP1` – `SWAP16` (Exchange Operations).
- 0xA0 - 0xA4: `LOG0` – `LOG4` (Logging Operations).
- 0xF0 - 0xFF: `CALL`, `RETURN`, `DELEGATECALL` (System Operations).

B DETAILS OF THE DATASET

Our objective in constructing the dataset is to collect a set of real-world Ethereum smart contracts, which can serve as a depository suite for security research on smart contracts. In particular, this dataset can be used to evaluate the effectiveness of smart contract vulnerability detection tools. Here, we further present the specific statistics of the dataset collection and labeling strategies.

Table 4 showcases the dataset statistics. Specifically, we have collected a total of 514, 880 functions from available 42, 910 Ethereum smart contracts with source code. In the 514, 880 functions, 701 functions of 680 contracts have the reentrancy (RE) vulnerability (*i.e.,* label = 1). 3, 368 functions of 2, 242 contracts possess the timestamp dependence (TD) vulnerability. Around 3, 503 functions of 1, 368 contracts have the integer overflow/underflow (IO) vulnerability. 149 functions of 136 contracts possess the delegatecall (DT) vulnerability.

Dataset Labeling. To facilitate data labeling, we refer to several *patterns* to filter out suspicious functions, which are then handed over to human experts for further manual verification. Taking the timestamp dependence vulnerability as an example, pattern **TDInvocation** models whether there exists an invocation to *block.timestamp* in a function. **TDAssign** checks whether the value of *block.timestamp* is assigned to other variables or passed to a condition statement as a parameter, and **TDContaminate** validates if *block.timestamp* may contaminate the triggering condition of critical operations. We consider a function as suspicious to have a timestamp dependence vulnerability if it fulfills the combined pattern: **TDInvocation** $\wedge$ (**TDAssign** $\vee$ **TDContaminate**). We then hand over the suspicious ones to peers with specialized knowledge for labeling. We have released the benchmark dataset on Github, where more details on the dataset and annotations can be found.

Table 4: Dataset Statistics. There are four types of vulnerabilities in the benchmark dataset.

Functions	RE. Functions	TD. Functions	IO. Functions	DT. Functions
514,880	701 / 2,505	3,368/ 6,285	3,503 / 10,774	149 / 436
Contracts	**RE. Contracts**	**TD. Contracts**	**IO. Contracts**	**DT. Contracts**
42,910	680 / 2,385	2,242 / 4,490	1,368 / 7,183	136 / 414

C TRAINING DETAILS OF BERT

It is worth mentioning that the BERT network is of several advantages [11]. (1) The in-depth pre-training task and fine-tuning task for the BERT contribute to yielding better semantic features. (2) The masked language task (MLM) and adjacency block prediction task (ABP) can help the BERT obtain both the instruction-level and the block-level information in the control flow graph (CFG). Inspired by the success of the BERT network in handling the program instructions, we resort to a pre-trained model BERT to deal with the bytecode basic blocks in the CFG. Empirically, in our work, the optimized BERT indeed has the ability to generate better semantic embeddings for the bytecode basic blocks.

Specifically, to pre-train a BERT network for processing the bytecode feature, we need to provide a pre-training strategy and an instruction training dataset. First, we construct a vocabulary, *i.e.*, *vocab*, to map the word piece (*i.e.*, instruction) into the unique id (*i.e.*, <ID, Word>) for the bytecode instructions. Then, we pre-train the BERT model using more than one million lines of instructions from scratch with following configurations: 1) *learning rate* = 2e-5, 2) *train steps* = 2,000, 3) *batch size* = 32, and 4) *max sequence length* = 64. Dimension for BERT pre-training embedding is 256.

Considering the discrepancy between different vulnerabilities, we design a dedicated fine-tuning task for the pre-trained BERT, from which we expect that the fine-tuned BERT is able to encode instructions with vulnerability characteristics more accurately. Moreover, using the fine-tuned BERT to extract the block embeddings of CFG could alleviate the noise in subsequent graph feature extraction. For each vulnerability, we enforce more than 100 thousand instructions for model fine-tuning with the default parameter settings: 1) *learning rate* = 2e-5, 2) *train epoch* = 10, 3) *batch size* = 32, and 4) *max sequence length* = 64. Note that, for each vulnerability, we have an independent fine-tuned BERT model. Finally, we exploit the fine-tuned BERT model to extract the semantic features of the bytecode basic blocks. As a side contribution, we have also released a pre-trained BERT model and four fine-tuned BERT models, hoping to facilitate community research.

D BROADER IMPACT

In this work, we proposed a novel cross-modality mutual learning framework to improve smart contract vulnerability detection on bytecode. To the best of our knowledge, this is the first work that investigates distilling knowledge from the teacher network for reconstructing the missing source code modality, which completes the bytecode modality towards more precise vulnerability detection. The attempt might inspire future research in this field.