Received February 15, 2022, accepted March 15, 2022, date of publication March 24, 2022, date of current version March 29, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3162065

CodeNet: Code-Targeted Convolutional Neural Network Architecture for Smart Contract Vulnerability Detection

SEON-JIN HWANG, SEOK-HWAN CHOI, JINMYEONG SHIN, AND YOON-HO CHOI, (Member, IEEE)
School of Computer Science and Engineering, Pusan National University, Busan 46241, Republic of Korea

Corresponding author: Yoon-Ho Choi (yhchoi@pusan.ac.kr)

This work was supported in part by Korea Electric Power Corporation under Grant R22XO01-3; and in part by the Ministry of Science and ICT (MSIT), South Korea, through the Information Technology Research Center (ITRC) Support Program supervised by the Institute of Information & Communications Technology Planning & Evaluation (IITP) under Grant IITP-2022-2020-0-01797.

ABSTRACT A smart contract is a computer program which is automatically executed with some conditional statements such as "if/then". Since smart contracts can include some vulnerable program codes, smart contract exploit was recently highlighted as one of the severe threats to Ethereum blockchain. As one of the efficient and effective smart contract vulnerability detection methods, deep learning methods have been studied due to the fast detection speed and the high detection accuracy. Recently, the deep learning methods using convolutional neural network(CNN) have actively studied to classify images transformed from smart contracts into vulnerable or invulnerable. However, while simply transforming a smart contract into an image and analyzing, semantics and context of the smart contract are ignored to cause false detection alarms. To detect vulnerable smart contracts while maintaining their semantics and context, we propose a new code-targeted CNN architecture, called CodeNet. To improve the performance of CodeNet, we also design a data pre-processing procedure, where a smart contract is transformed into an image while maintaining locality. From the experimental results under various types of vulnerabilities, the proposed CodeNet-based vulnerability detection method shows the good-enough detection performance and detection time compared to well-known state-of-the-art vulnerability detection tools.

INDEX TERMS Blockchain, convolutional neural network, deep learning, Ethereum, smart contract, vulnerability detection.

I. INTRODUCTION

The Ethereum platform [2], or simply called Ethereum, is a platform powered by blockchain technology. The Ethereum platform works using three key elements of blockchain technology, i.e, distributed ledger technology with immutable records, consensus algorithm and smart contract. While distributed ledger technology and consensus algorithm make the Ethereum platform secure, smart contracts make the Ethereum platform execute the actions to govern associated assets. Many decentralized finance(DeFi) and other applications such as logistics, insurance, medicine, energy, Non-Fungible Token(NFT), and so on, use smart contracts.

The associate editor coordinating the review of this manuscript and approving it for publication was Haiyong Zheng.

A smart contract is a computer program, which automatically executes the actions when predetermined conditions following *"if…then…"* statements in a smart contract are met. Typically, the Ethereum smart contract is written in a Turing-complete programming language, called Solidity [20]. Since Solidity uses strongly-typed variable declaration and has some conditional statements, new smart contract vulnerabilities such as *unchecked low level(LL) calls*, *timestamp dependency*, and so on are found and exploited by malicious users. For example, the decentralized autonomous organizations(DAO)'s smart contract vulnerability due to *reentrancy* vulnerability caused a loss of over 150 million dollars [5] in 2016.

With automated execution and public visibility, smart contract can provide strong reliability without an authorized third party. Since smart contracts are distributed within the

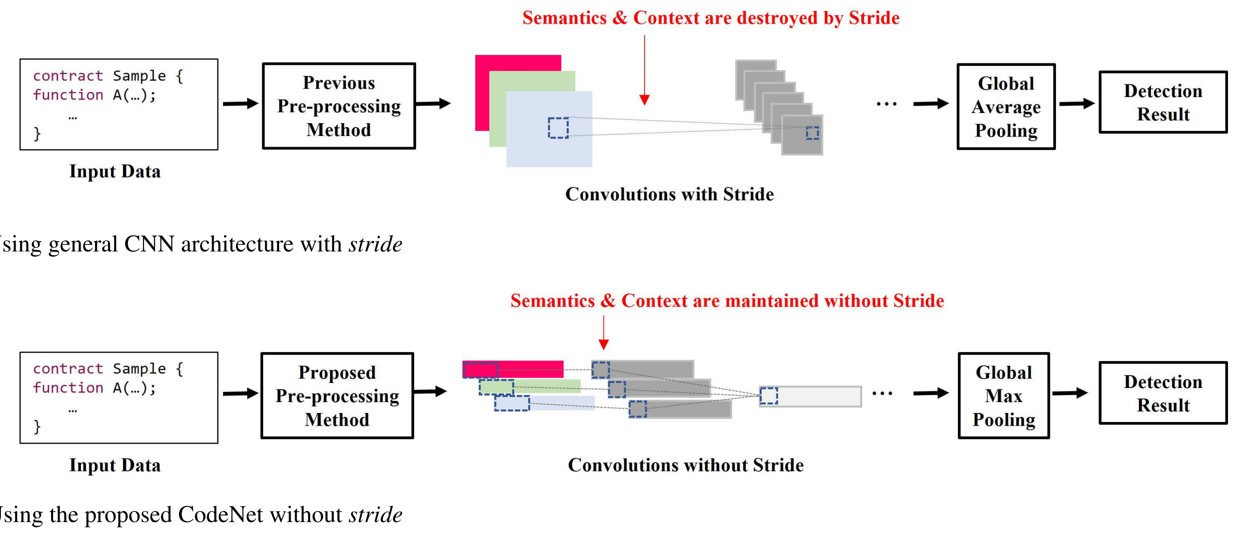

Semantics & Context are destroyed by Stride
contract Sample {
function A(…);
…
}
Input Data
Previous Pre-processing Method
Convolutions with Stride
Global Average Pooling
Detection Result
(a) Using general CNN architecture with stride
Semantics & Context are maintained without Stride
Proposed Pre-processing Method
Convolutions without Stride
Global Max Pooling
(b) Using the proposed CodeNet without stride

FIGURE 1. **Overall operational comparison for smart contract vulnerability detection using CNN architectures.**

blockchain network and are publicly visible to everyone, unintended bugs and vulnerabilities are also visible to everyone. Smart contract vulnerability exploits are serious because vulnerabilities in the deployed smart contracts cannot be corrected due to the immutable nature of the distributed ledger. Due to these characteristics, an attacker can easily exploit smart contract bytecodes after identifying whether the vulnerability is present or not. If a smart contract vulnerability is identified, the attacker can exploit the smart contract to leak digital assets.

To prevent the vulnerability exploits, smart contract developer should check potentially vulnerable codes before deploying smart contracts according to secure coding guidelines, called security patterns. Security patterns of smart contract can provide a reliable contract execution to mitigate losses of digital assets [31]. Note that defining security patterns requires deep knowledge about smart contract execution because developers need to consume the significant efforts while identifying vulnerable codes. To resolve such a problem, automated vulnerability detection tools are developed to detect vulnerabilities on smart contracts before deploying smart contracts on the blockchain network.

Automated vulnerability detection tools based on symbolic execution and non-symbolic execution have typically been used due to its high detection accuracy for known vulnerabilities. However, symbolic execution tools such as Oyente [17], Osiris [28], Mythril [6], and Securify [29] are time-consuming at simulating symbolic paths and are not suitable for batch vulnerability detection because it is difficult to explore all executable paths in a contract. Non-symbolic execution tools such as Slither [8] and Smartcheck [27] are also time-consuming at simulating symbolic paths and cause false negatives because they highly depend on the predefined detection rules.

Compared to other static analysis methods, deep learning methods show good vulnerabilities detection accuracy without explicit definition of the detection rules from training datasets [30]. Also, deep learning methods provide constant execution time by calculating the product of learned weight values and input values from the smart contract. Especially, convolutional neural network(CNN)-based detection methods have generally shown good accuracy when classifying malware and vulnerable software codes [13], [21]. However, since CNN-based deep learning models [22], [26] are designed to classify not software codes but images, they cannot correctly analyze semantics and context of smart contracts to cause many false positives and false negatives.

As shown in Fig. 1a, vulnerability detection methods using general CNN architectures such as Inception [26] and VGGNet [22] simply classify smart contract vulnerabilities using 2-D normal convolution with *stride*. Here, *stride* are required to downsample features in feature maps while shifting the input matrix in a convolutional neural network. When downsampling features in feature maps, *stride* operations can destroy semantics and context of the original smart contract. For example, let us assume that an instruction in a smart contract consists of a sequence of six bytecodes, i.e., push(0×63) with operands 0xaabbccdd and pop(0×50) with no operand. To analyze the instruction using CNN, the sequence of bytecodes is transformed into a sequence of six pixels on a smart contract image. If a *stride* of size 3 is applied into the sequence of pixels, six pixels are reduced into only two pixels. Such downsampling means that a group of bytecodes, 0xaabb and push(0×63), and another group of bytecodes, 0xccdd and pop(0×50), are mapped into each pixel. As a result, the execution result of the instruction is different from the original one.

In this paper, to ensure the good-enough performance of CNN-based vulnerability detection method, we propose a new code-targeted CNN architecture, called CodeNet. Different from the well-known CNN architectures using *stride*, CodeNet is designed to detect vulnerable smart contracts without *stride* while maintaining semantics and context of smart contracts as shown in Fig. 1b. To detect smart contract vulnerabilities using CNN effectively, CodeNet operates following a data pre-processing procedure, which makes an input image for CodeNet to be generated while maintaining the semantics and context of the original smart contract.

Main contributions of this paper can be summarized as follows: (1) We propose a new CNN architecture to classify vulnerable smart contracts while maintaining semantics and context of the smart contract. Different from other CNN architectures using *stride*, we design a new CNN architecture without *stride*; (2) We propose an effective data pre-processing method which maintains locality of the smart contract for CodeNet. While converting smart contracts into input images for CNN, we maintain semantics and context of a smart contract; (3) From experimental evaluation results, we show that the proposed CodeNet-based vulnerability detection method provides the good-enough performance compared to six state-of-the-art vulnerability detection tools.

The rest of the paper is organized as follows. In section II, we overview well-known smart contract vulnerabilities and describe the related works. We show the operation of the proposed CodeNet-based vulnerability detection method, which consists of a data pre-processing step and a vulnerability detection step, in section III. In section IV, we show the evaluation results of the proposed CodeNet-based vulnerability detection method under various experiments. Finally, we conclude this paper in section V.

II. BACKGROUND & RELATED WORK

To understand smart contract vulnerabilities, we overview the characteristics of well-known smart contract vulnerabilities. Discussion on smart contract vulnerabilities is confined to the Ethereum smart contracts programmed using Solidity. Also, to understand how to detect smart contract vulnerabilities, we overview state-of-the-art automated vulnerability detection tools. Since most of automated vulnerability detection tools analyze smart contract vulnerabilities using the static analysis method, discussion mainly focuses on symbolic execution and non-symbolic execution methods.

A. SMART CONTRACT VULNERABILITY

Reentrancy vulnerability is a vulnerability caused by the fallback function of the smart contract. The fallback function is called when a smart contract(**Alice**) receives *ether* from another contract(**Bob**) using `call.value` function. If **Bob** receives *ether* from **Alice** and **Bob** writes `send` function in fallback function, *reentrancy* vulnerability can exist. Actually, *reentrancy* vulnerability was found and exploited in a DAO smart contract to cause the loss of millions of dollars in June 2016 [5].

Unchecked Low Level Calls vulnerability can exist when using low level functions such as `send`, `call`, `callcode` and `delegetecall`. Since such low level functions do not revert previous executions when the function execution fails, such a vulnerability exploit may cause an unexpected side effect that *ether* decreased but is not sent.

Timestamp Dependency vulnerability is related to the `block.timestamp` variable. The `block.timestamp` variable is a global variable that contains timestamp of the current block as seconds since the Unix epoch. This timestamp value is generally used for random value generation in blockchain. Since blockchain is a public ledger which everyone can use, the random value should be determined by the time when everyone can accept. The only value that satisfies the condition is `block.timestamp`. It is determined when a block is mined. However, since a miner can determine the time when a block is mined, the miner can control the random value.

Tx.origin vulnerability is closely related to the social engineering attack. In the smart contract, `tx.origin` contains the address of transaction creator and is used to confirm that the function caller is a valid user for some important functions. Since `tx.origin` only contains transaction creator's information even if an important function is called by another smart contract, the important function cannot recognize whether being called in unintended way. Thus, an adversary can phish a victim to call important functions using malicious smart contract codes.

B. SYMBOLIC EXECUTION

Even though the performance is limited because of high false negative rate [7], much time consumption while exploring all possible program paths, and much emulation time for symbolic paths, symbolic execution methods have widely implemented in well-known smart contract vulnerability detection tools.

Oyente Luu, *et al.* [17] proposed a smart contract vulnerability detection method, called Oyente, which is based on control flow graph(CFG) construction and symbolic execution. Oyente used two inputs including bytecode of a contract and the current Ethereum global state. After constructing control flow graph using the bytecode and CFGBuilder, Oyente runs symbolic execution with Explorer. Finally, CoreAnalysis components identify the output value. However, high false positive rates due to low code coverage limited the performance of Oyente.

Osiris Christof Ferreira Torres *et al.* [28] proposed Osiris, which is a framework on the top of Oyente's symbolic execution engine. They combined symbolic execution with taint analysis. They constructed a CFG and forwarded every executed instruction to the taint analysis component. The taint analysis component verified that the executed instruction was a part in the list of defined sources. Being compared to Oyente, Osiris showed outstanding performance for the integer bugs vulnerability. However, the performance of Oyente is still limited by high false positive rate due to low code coverage.

Maian Nikoli *et al.* [19] proposed a dynamic analysis method, called Maian, which extended Oyente. To detect smart contract vulnerabilities, Maian created blockchain transactions in the private blockchain and used them as inputs for symbolic execution analysis. Maian used systematic techniques to find two violation properties of traces, i.e., safety properties and liveness properties. Maian asserts that there exists a trace from a specified blockchain state, which causes the contract to violate certain conditions for safety properties, and that some actions cannot be taken in any execution starting from a specified blockchain state in liveness properties. However, high false positive rates due to low code coverage limited the performance of Maian.

Mythril [6] To find the symbolic execution paths and improve code coverage that cause the smart contract vulnerabilities, there have been several studies. Mythril, developed by ConsenSys, used symbolic execution using LASER-Ethereum and taint analysis to detect vulnerabilities. They modeled smart contract execution as a space of states and used path formulas in propositional logic. They produced statements using the logic and identified those statements in the context of the model.

Securify Tsankov *et al.* [29] proposed Securify, which is the abstract interpreter applied to smart contract vulnerability detection, to provide soundness guarantees over all possible executions. They symbolically analyzed smart contract dependency graph and extracted semantic information from the contract. Next, they checked vulnerable patterns that satisfied conditions. Securify showed lower false negative rate than Oyente and Mythril.

Manticore Mossberg *et al.* [18] proposed a detection method using dynamic symbolic execution. Using emulation, they support both traditional computing environments (x86/64, ARM) and exotic ones, such as the Ethereum platform. They emulated environment for smart contract execution to support an arbitrary number of interacting contracts. From the experimental results using symbolic emulation, they show that the proposed dynamic symbolic execution method [18] can remove false positives in the symbolic path and improve the code coverage.

EOSAFE He *et al.* [12] presented EOSAFE for EOSIO, one of the representative Delegated Proof-of-Stake (DPoS) blockchain platforms. As the first static analysis framework to automatically detect vulnerabilities in EOSIO smart contracts, They used a practical symbolic execution engine for WebAssembly(Wasm) bytecode and a customized library emulator, and heuristic-driven detectors to identify four vulnerabilities in EOSIO smart contracts.

RA Chinen *et al.* [3] presented a *reentrancy* vulnerability analysis tool, called RA (Re-entrancy Analyzer). After combining symbolic execution and equivalence checking methods, they showed that *reentrancy* vulnerabilities were well verified without prior knowledge of attack patterns and spending Ether.

C. NON-SYMBOLIC EXECUTION

Slither Feist *et al.* [8] proposed a static analysis framework, called Slither, to provide rich information about smart contracts. They generated an inheritance graph and CFG from the abstract syntax tree(AST) generated from the smart contract source code. Slither converts the entire smart contract source code into an internal representation language, called SlithIR. Next, Slither detects smart contract vulnerabilities while comparing source codes with a set of predefined detection rules. However, since static detection tools without symbolic execution are highly dependent on the predefined detection rules, the performance of Slither is limited by false negatives.

Ethainter Brent *et al.* [1] proposed security analyzer, called Ethainter, for detecting composite information flow violations in Ethereum smart contracts. For the targeted composite vulnerabilities that escalate a weakness through multiple transactions, Ethainter show the low false positive rate.

MadMax Grech *et al.* [10] proposed MadMax, for finding so-called *gas*-focused vulnerabilities in Ethereum smart contracts. They used a static program analysis technique that automatically detects *gas*-focused vulnerabilities using a smart contract decompiler and semantic queries in datalog. They effectively identified a permanent denial-of-service attack on the contract which are hard for programmers to find.

VERISMART So *et al.* [25] presented VERISMART, which ensures arithmetic safety of Ethereum smart contracts. To detect arithmetic bugs, they implemented CEGIS-style algorithm in a tool that leverages transaction invariants automatically during the verification process. Using real-world smart contracts, the authors showed that VERISMART is effective at detecting arithmetic bugs.

Reguard Liu *et al.* [16] designed Reguard which converts the smart contract source code into an intermediate form to automatically detect reentrancy bugs in the Ethereum smart contracts. ReGuard transforms the converted intermediate representation(IR)s into C++ codes and uses them as inputs of fuzzing engine. While analyzing runtime traces generated by fuzzing engine, Reguard causes false negatives depending on the performance of fuzzing engine [7].

Contractfuzzer Jiang *et al.* [14] proposed a novel fuzzer, called Contractfuzzer, to test smart contract vulnerabilities. Contractfuzzer analyzes the smart contract vulnerability by using both the online module and the offline module. In the offline module, Contractfuzzer collects application binary interface(ABI) for the smart contract and binary code in Etherscan. The collected contracts are distributed to the private network built in advance and used for fuzzing. In online module, Contractfuzzer analyzes the collected contract ABI and binary code, and generates fuzzing input. By using the fuzzing input, Contractfuzzer gets execution log files which contain call stack conditions. The execution log files are used for vulnerabilities detection. Such a dynamic analysis detection method shows low false positive rates because vulnerabilities are found from execution results. However, the performance of Contractfuzzer is highly dependent on the performance of fuzzer and is limited due to much time consumption while emulating inputs [7].

Contractward Wang *et al.* [30] proposed a machine learning-based smart contract vulnerability detection method, called ContractWard. In order to reduce the dimensionality

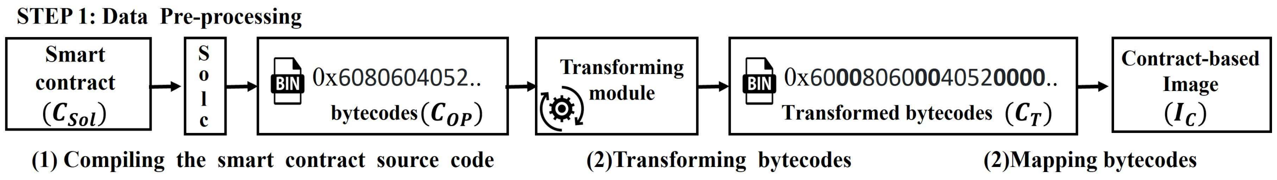

STEP 1: Data Pre-processing
Smart contract (C_{Sol})
Solc
BIN 0x6080604052..
bytecodes (C_{OP})
Transforming module
BIN 0x6000806000405200000..
Transformed bytecodes (C_T)
Contract-based Image (I_C)
(1) Compiling the smart contract source code
(2)Transforming bytecodes
(2)Mapping bytecodes

FIGURE 2. Overall operation of the proposed data-preprocessing step.

of the features, the authors classified functionally similar opcodes into one category. For example, e.g., PUSH1, PUSH32 and so on, are classified into a PUSH opcode. By using features with the reduced dimension, the authors evaluated the detection performance of various machine learning-based classification algorithms such as eXtreme Gradient Boosting(XGBoost), Adaptive Boosting(AdaBoost) and so on. Since being designed using a CNN model, the proposed CodeNet-based vulnerability detection method is similar to Wang *et al.*'s work. However, the proposed CodeNet-based vulnerability detection method is different from Contractward since we propose a new CNN architecture to improve the performance of a CNN-based smart contract vulnerability detection method.

III. PROPOSED METHOD

In this section, we describe the overall operation of the proposed CNN-based vulnerability method. We also explain the detailed operation of the proposed CNN architecture, called CodeNet, for effective and efficient smart contract vulnerability detection. Overall operation of the proposed CodeNet-based vulnerability detection method consists of two steps as follows:

- **Step 1. Data Pre-process:** After the smart contract source code is compiled into bytecodes, bytecodes are transformed into a smart contract-based input image for CNN architectures while keeping semantics and context of a smart contract. (Lines 2 to 4 in Algorithm 1)
- **Step 2. Vulnerability Detection:** To detect vulnerable smart contracts, the proposed CodeNet-based vulnerability detection method analyzes the smart contract-based input images. (Line 5 in Algorithm 1)

Algorithm 1 Proposed Method Algorithm

```
    Input: $\mathbf{C}_{Sol}$: Smart contract source code
    Output: R : Detection result
1:  procedure Proposed method($\mathbf{C}_{Sol}$)
2:      $\mathbf{C}_{OP}$ = Compile($\mathbf{C}_{Sol}$)
3:      $\mathbf{C}_{T}$ = Transform($\mathbf{C}_{OP}$)
4:      $\mathbf{I}_{C}$ = Image_Mapping($\mathbf{C}_{T}$)
5:      $\mathbf{R}$ = Predict($\mathbf{I}_{C}$)
6:      return R
7:  end procedure
```

In the data pre-processing step in lines 2-4, the smart contract source code is transformed $\mathbf{C}_{Sol}$ into contract-based image $\mathbf{I}_C$. After compiling the smart contract source code $\mathbf{C}_{Sol}$ to bytecodes $\mathbf{C}_{OP}$ in line 2, the pre-processing module transforms bytecodes $\mathbf{C}_{OP}$ into a fixed size of codes $\mathbf{C}_T$ in line 3. To generate an input image for CodeNet, the data pre-processing module converts a fixed size of codes $\mathbf{C}_T$ into a contract-based image $\mathbf{I}_C$ in line 4. In the vulnerability detection step in line 5, CodeNet identifies whether the contract-based image $\mathbf{I}_C$ is vulnerable or not. Details of each step are shown in the followings.

A. DATA PRE-PROCESS

In the data pre-processing step, the data pre-processing module transforms the smart contract into the contract-based image following of three functional procedures as shown in Fig. 2: (1) Compiling the smart contract source code; (2) Transforming bytecodes into a fixed size of code; and (3) Mapping bytecodes to a contract-based image, which is an input image for training CodeNet.

1) COMPILING SMART CONTRACT SOURCE CODE

To train CNN architectures using the smart contract source code, we compile a high-level source code to bytecodes using a compiler. Since we consider solidity-based smart contract codes, *solc* compiler is used to compile the Ethereum smart contract source code.

2) BYTECODE TRANSFORMATION

Since the convolution operation of CNN takes fixed size convolution filter, CNN does not describe an opcode and an operand. For example, let us consider bytecodes, which is a part of smart contract, 0×60806240507852. In the Ethereum operation, 0×60 is PUSH1 opcode that takes a single argument as next bytecode. Next bytecode, 0×80, is operand for 0×60. That is, 0×6080 indicate one instruction that pushes 0×80 to the stack. Bytecodes, 0×60806240507852, are analyzed in the Ethereum virtual machine(EVM). EVM takes instruction PUSH1(0×60) opcode with argument 0×80, PUSH3(0×62) opcode with argument 0×405078, and MSTORE(0×52) opcode with no argument. However, if the size of convolution filter is 1×3 with *stride* 3, 0×608062, 0×405078 and 0×520000 are analyzed with convolution filter. Even though 0×6080, 0×62405078 and 0×52 should be binded, CNN simply analyzes bytecodes using fixed size 1×3 of convolution filter without considering code semantics.

To perform the convolution operation with a fixed size convolution filter, the pre-processing module transforms bytecodes into a fixed-size code. Note that bytecodes consist of the Solidity opcodes and arguments for each instruction. To get the fixed-size codes from bytecodes, we adjust the number of arguments of each instruction. Even though most Solidity instruction does not take arguments directly in bytecodes [32], PUSH opcode takes 1 to 32 number of arguments. That is, it is possible to get the fixed-size codes by adjusting the number of arguments for PUSH opcode in bytecodes.

We determine the reasonable number of arguments to get the fixed-size code from bytecodes after analyzing PUSH opcode arguments in 29,022 number of smart contracts. As shown in Fig. 3, we found 48,069,183 number of PUSH opcodes in total and estimated their distributions for different numbers of arguments. We observed that the ratio of PUSH opcodes with one or two arguments was 78.8% of the total. Thus, we determined the number of arguments into two to get a fixed-size code from bytecodes. For PUSH opcode with three or more arguments, we only used first two arguments and removed other arguments. For other opcodes without arguments or with only one argument, we added zero padding.

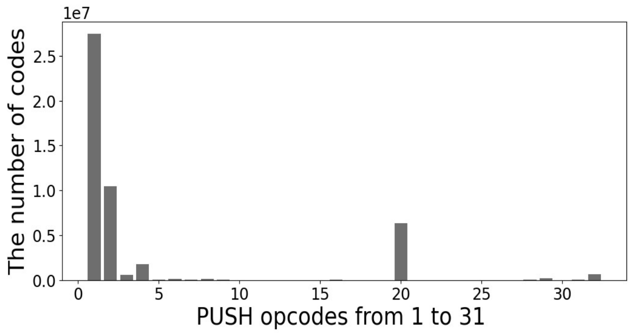


FIGURE 3. PUSH instructions distribution in the 29,022 smart contract.

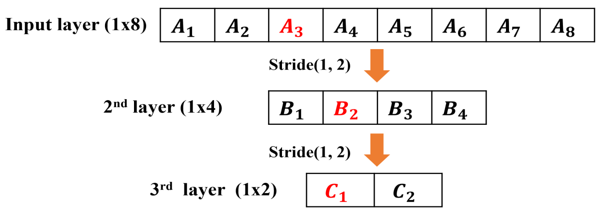
For example, let us consider bytecodes, 0×60806240507852, which is a part of a smart contract. Since 0×60 is PUSH1 opcode that takes one argument as a next bytecode, the next bytecode 0×80 is operand for PUSH1(0×60). That is, 0×6080 indicates one instruction that push 0×80 to the stack. However, since 0×60 takes only one argument, one zero padding is added to 0×6080. As a result, 0×6080 is transformed into 0×608000. The bytecode 0×62 indicates PUSH3 opcode and takes three arguments. That is, even though the next 0×40, 0×50, and 0×78 bytecodes are operands for 0×62, 0×78 in bytecodes is ignored since the number of arguments while getting the fixed-size code is two. Also, since 0×52 in bytecodes indicates MSTORE opcode, which takes no argument, two zero paddings are added following 0×52 in bytecodes.

3) MAPPING BYTECODE TO CONTRACT-BASED IMAGE

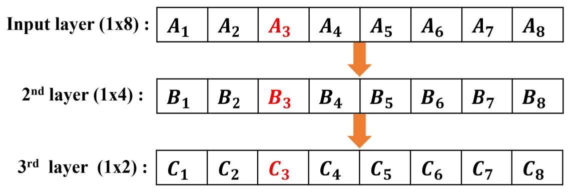
To generate an input for CNN architectures, the proposed pre-processing module converts the fixed-size code to an RGB image. Since both the fixed-size code and the RGB image are expressed into hexadecimal form, the fixed-size code can be easily mapped to the RGB image. For example, let us consider a sequence of bytecodes, 0×608000604050. A part of bytecodes with the size 3 is mapped to an RGB pixel. That is, one pixel with R: 0×60, G: 0×80, B: 0×00 and another pixel with R: 0×60, G: 0×40, B: 0×50. Such a mapping is important because after converting the fixed-size code to an RGB image, CodeNet adjusts 2-D RGB image to 1-D RGB image to preserve semantics and context of bytecodes. For example, 100×100x3 RGB image is adjusted to 1×10000x3 because 2-D operation is not validated in the opcode sequence.

B. VULNERABILITY DETECTION

Since semantics and context of the natural image are not greatly influenced by independent pixels, the state-of-the-art CNN architectures [11], [22], [26] showed the good performance when considering the color locality of image pixels. However, the performance of the state-of-the-art CNN architectures is limited because convolution with *stride* vanishes semantics and context of a smart contract while analyzing dependent pixels. Intuitively, *stride* make CNN architectures consider higher-level features rather than pixel-by-pixel features. Note that the higher-level feature is extracted, the more pixel-level features are lost. As a result, semantics and context of the smart contract are destroyed [13] because a pixel of contract-based image represents instructions with opcodes and operands. For example, let us assume that the size of input is 1×8 and the size of *stride* is 1×2 as shown in Fig. 4a. Here, A_i, B_i, and C_i are feature map values in different layers and i indicates the position of a feature. A_3 in the input layer is



(a) Local information vanishment following feature mapping of CNN architecture with strides


Input layer (1x8) :
A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8
2nd layer (1x4) :
B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8
3rd layer (1x2) :
C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8

(b) Local information maintenance following feature mapping of non-strides CNN architecture

FIGURE 4. Local information variance from feature mapping of CNN architectures with strides and without strides.

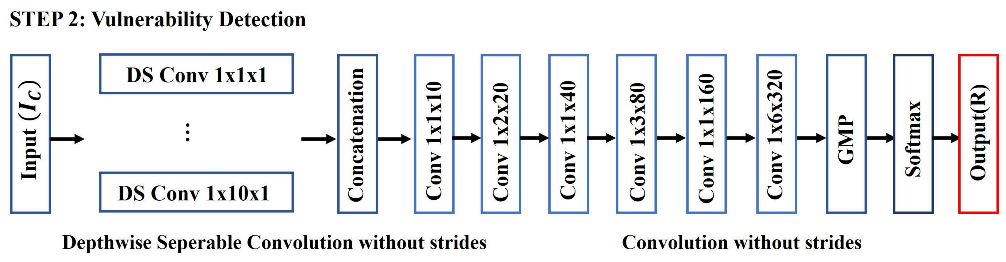


FIGURE 5. **The proposed CodeNet model architecture.**

mapped into B_2 in the 2nd layer. B_2 in the 2nd layer is mapped into C_1 in the 3rd layer. As a result, the vulnerability classification results of the CNN-based vulnerability detection methods become inaccurate because the local information including code sequences are lost. To solve the problem mentioned above, we propose a new CNN architecture, called CodeNet, for smart contract vulnerability detection while maintaining semantics and context of a smart contract. The proposed CodeNet model architecture is shown in Fig. 5.

In order to maintain semantics and context of the original smart contract, CodeNet does not use *stride*. For example, let us assume that the size of input is 1×8 as shown in Fig. 4b. A_3 in the input layer is mapped into B_3 in the 2nd layer. Next, B_3 in the 2nd layer is mapped into C_3 in the 3rd layer. Even though the *stride* operation does not change the local information, feature map values change due to convolution without *stride* but do not vanish. As a result, the vulnerability classification result of the CodeNet-based vulnerability detection method does not change because the local information including code sequences does not change. However, since a non-*stride* CNN architecture maintains all features in each layer, the performance of non-*stride* CNN architecture is limited due to large amount of computation and lack of memory due to many parameters.

In order to overcome large amount of computation and lack of memory due to many parameters, CodeNet is designed using depthwise separable(DS) convolutions and non-*stride* convolutions as shown in Fig 5. The depthwise separable convolutions technique [4] consists of two types of convolutions, i.e., depthwise convolution and pointwise convolution. Different from the original convolution where the number of multiplication increases according to the number of input channels, depthwise convolution uses a kernel which has the same number of channels as the input. Thus, the dimension of the output is always the same as the dimension of the input. The pointwise convolution uses a 1×1 kernel which projects the channels output by the depthwise convolution onto a new channel space. As a result, by applying pointwise convolution following depthwise convolution [4], i.e., the DS convolution, we get the output which has the same dimension as that of the original convolution with *stride*. From the empirical analysis results, we observed that even though the number of parameters decreased more than 80%, the accuracy of depthwise separable convolution was similar

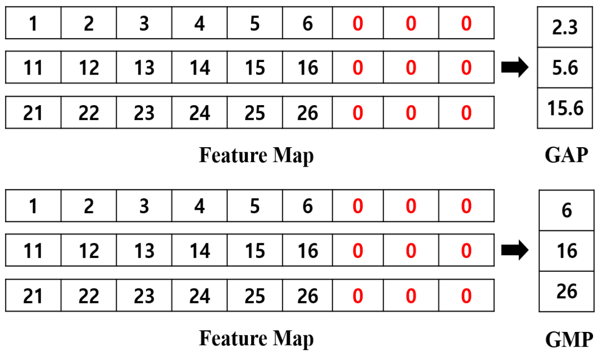


FIGURE 6. **Comparison between GAP and GMP.**

to that of the original convolution. To detect smart contract vulnerabilities while maintaining semantics and context of a smart contract, we design CodeNet using depthwise separable convolution [4] following the input layer.

Let us note that $1 \times n$ size of filter extracts features from n number of instructions. For example, 1×1 and 1×2 sizes of filters extracts features from 1 and 2 numbers of instructions respectively. In order to extract n number of features from instruction sequences in a smart contract, different sizes of filters in DS convolution layers should be used. Thus, CodeNet is designed using 10 number of customized filters, whose sizes are different respectively. To extract features from different numbers of instruction sequences, 10 number of output feature maps from every customized filters are concatenated as a single feature map. As a result, instruction sequences with different sizes can be analyzed as inputs for vulnerability detection without information loss. Next, the concatenated output is operated using a convolution without *stride*, which transforms an image by applying a kernel over each pixel and its local neighbors while maintaining semantics and context of the smart contract image. Output across 6 number of convolution layers is now fed into a global max pooling(GMP) layer followed by a softmax layer for classification. Different from global average pooling(GAP), which takes the average of the feature map, GMP takes the maximum of the feature map. Thus, we selected GMP not to lose the original information in the zero-padded feature map. For example, as shown in Fig. 6, the GAP results for zero-padded feature map are 2.3, 5.6 and 15.6 while the

GAP results for feature map without zero padding are 3.5, 13.5 and 23.5. That is, the GAP results for feature map with zero padding lose feature values in the original feature map. Different from the GAP results, the GMP results are the same regardless of whether padding exists or not.

IV. EVALUATION RESULTS

A. EXPERIMENTAL ENVIRONMENTS

In this section, we show the experimental results for the vulnerability detection accuracy of the proposed CodeNet. We measured the detection accuracy and detection time of the proposed CodeNet-based vulnerability detection method for four types of vulnerabilities [7]. To show the vulnerability detection effectiveness of CodeNet, we compared the detection performance of the proposed CodeNet-based vulnerability detection method with state-of-the-art smart contract vulnerability detection tools. To show the vulnerability detection effectiveness of the proposed CodeNet, we compared the detection performance of CodeNet with state-of-the-art CNN architectures [22], [26]. To show the vulnerability detection efficiency of the proposed CodeNet-based vulnerability detection method, we compared the vulnerability detection time of the proposed one with state-of-the-art smart contract vulnerability detection tools.

We implemented the proposed CodeNet-based vulnerability detection method using TensorFlow-gpu version 2.2.0, Python version 3.6.9, and Keras version 2.4.2. For the efficient experiments, we measured the performance on the Ubuntu 18.04.1 LTS machine with kernel version 4.15.0-36-generic, AMD EPYC 7301 16-Core Processor, Nvidia RTX 2080 Ti and 128GB memory.

As datasets for training the proposed CodeNet, we used smartbug^{wild} [23] dataset which consists of 47,518 real-world contracts. To label vulnerable dataset, we used SolidiFI [9] which is a bug evaluation tool using 155 code snippets collected from SolidiFI github, Ethereum community, solidity official document and etc. While labeling invulnerable dataset, we ruled out contracts which contain potentially vulnerable characteristics. Such potentially vulnerable characteristics for four types of vulnerabilities are determined as follows [7].

- **Reentrancy.** Contracts which contain `call.value` function are potentially vulnerable to reentrancy vulnerability.
- **Unchecked Low Level Calls.** Contracts which contain `end`, `call`, `callcode` and `delegetecall` function can cause unchecked low level calls vulnerability.
- **Tx.origin.** Contracts which contain `tx.origin` variable potentially have Tx.origin vulnerability.
- **Timestamp Dependency.** Contracts which contain `block.timestamp` and `now` variables potentially have timestamp dependency vulnerability.

As a result, large sets of vulnerable data and invulnerable data for each type of vulnerability are collected as shown in Table 1. From each type of vulnerability dataset, we randomly selected about 70% of data into a train dataset, 20% data into a validation dataset, and the remaining data into a test dataset. To avoid false classification due to the invalid data, we excluded data with compile errors, solidity version errors, and so on.

TABLE 1. Total number of data for four types of vulnerability.

Categories	Vulnerable	Invulnerable
Reentrancy	24367	24447
Unchecked LL Calls	21730	20054
Tx.Origin	24374	26215
Timestamp Dependency	24473	17532

We measured performance of the proposed CodeNet-based vulnerability detection method using two datasets, i.e., smartbug^{wild} dataset and smartbug^{curated} [24]. Here, Smartbug^{wild} dataset consists of 13,443 smart contracts with four types of vulnerabilities and Smartbug^{curated} consists of 79 smart contracts for three types of vulnerabilities except for *tx.origin* vulnerability.

While measuring the detection accuracy, we compared the proposed method with benchmark tools tested on Smartbugs [7]. Smartbugs is a publicly available framework which provides nine benchmark tools to analyze a Solidity smart contract vulnerability. Among nine benchmark tools, we excluded three tools that showed 0% accuracy for target vulnerabilities. Finally, six state-of-the art tools named Mythril [6], Osiris [28], Oyente [17], Securify [29], Slither [8], and Smartcheck [27] are compared with the proposed CodeNet.

B. VULNERABILITY DETECTION EFFECTIVENESS

To compare the performance of the proposed CodeNet-based vulnerability detection method with other state-of-the-art tools, we measured the *accuracy*, *precision*, *recall*, and *f1-score* for vulnerability detection. In Table 2, we show the evaluation result of the proposed CodeNet-based vulnerability detection method compared with state-of-the-art tools for smartbug^{wild} dataset. Since Osiris and Oyente do not support *unchecked low-level calls* and *tx.origin* vulnerabilities, detection results for these two vulnerabilities are not available. Detection results of Securify and Mythril for the *timestamp dependency* vulnerability are not available because Securify and Mythril do not detect the *timestamp dependency* vulnerability.

Since six state-of-the-art tools use static rule to detect vulnerable smart contracts, false positive rates of these tools are low. Thus, *precision* values of six state-of-the-art tools are relatively higher than the other performance metrics. Compared to other vulnerabilities, most of detection tools showed the best detection performance for *reentrancy* vulnerability except Mythril. That is, Oyente, Slither, Smartcheck and the proposed CodeNet-based vulnerability detection method showed the good performance in *reentrancy* vulnerability detection. Next, while Smarcheck showed the

TABLE 2. Vulnerability detection accuracy, precision, recall and f1-score results comparison to static analysis tools on smartbug^{wild}.

Vulnerability	Evaluation Metric	Tool						
		Mythril [6]	Osiris [28]	Oyente [17]	Securify [29]	Slither [8]	Smartcheck [27]	CodeNet
Reentrancy	Accuracy	69.02%	75.42%	95.65%	82.71%	93.51%	**99.91%**	98.27%
	Precision	79.14%	**100%**	**100%**	94.17%	93.20%	99.81%	97.65%
	Recall	49.69%	49.69%	91.10%	68.87%	93.54%	**100%**	98.84%
	F1-score	61.05%	66.39%	95.34%	79.56%	93.37%	**99.90%**	98.24%
Unchecked LL Calls	Accuracy	71.45%	N/A	N/A	84.16%	77.80%	88.64%	**98.79%**
	Precision	99.85%	N/A	N/A	**100%**	87.14%	98.89%	99.24%
	Recall	41.36%	N/A	N/A	67.43%	63.77%	77.52%	**98.26%**
	F1-score	58.49%	N/A	N/A	80.54%	73.65%	86.91%	**98.75%**
Tx.origin	Accuracy	73.07%	N/A	N/A	82.49%	94.73%	**100%**	99.21%
	Precision	97.46%	N/A	N/A	99.62%	95.75%	**100%**	99.87%
	Recall	46.94%	N/A	N/A	64.95%	93.52%	**100%**	98.53%
	F1-score	63.37%	N/A	N/A	78.64%	94.62%	**100%**	99.20%
Timestamp Dependency	Accuracy	N/A	56.72%	53.54%	N/A	87.16%	75.51%	**94.35%**
	Precision	N/A	97.54%	88.13%	N/A	97.18%	**100%**	90.40%
	Recall	N/A	9.62%	3.14%	N/A	75.30%	48.72%	**98.66%**
	F1-score	N/A	17.52%	6.07%	N/A	84.85%	65.52%	**94.35%**

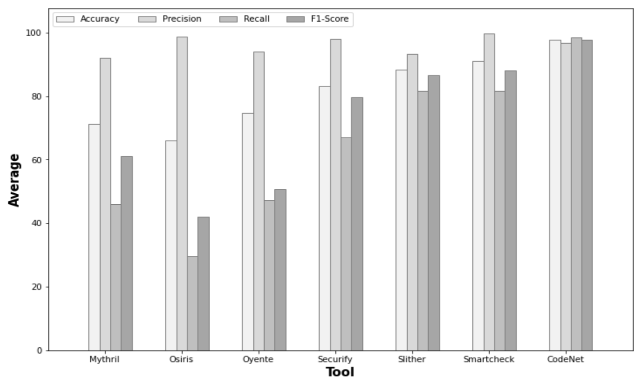


FIGURE 7. Average performance comparison of six well-known tools and the proposed CodeNet on smartbug^{wild} dataset.

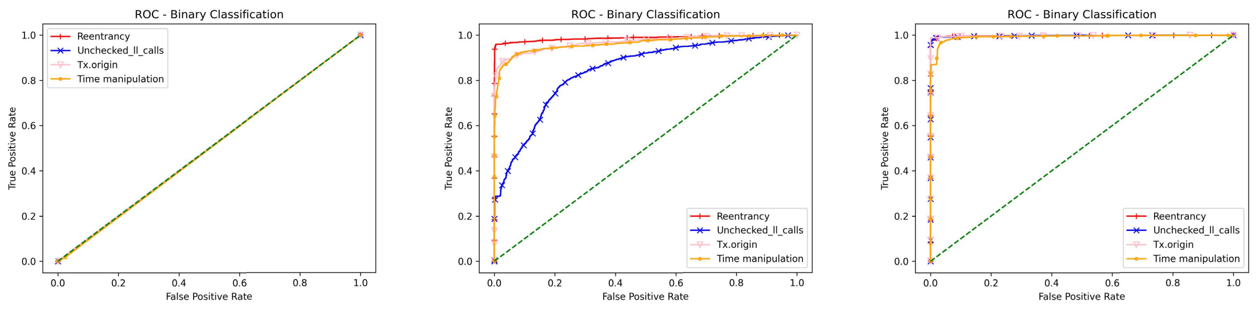
best performance for *reentrancy* and *tx.origin* vulnerabilities, the proposed CodeNet-based vulnerability detection method showed better performance than Oyente and Slither. In *unchecked low-level calls* vulnerability detection, the proposed CodeNet-based vulnerability detection method showed 98.79% for *accuracy*, 98.26% for *recall* and 98.75% for *f1-score*, which are the highest values. While Smartcheck showed 100% of performance in *tx.origin* vulnerability detection, the proposed CodeNet-based vulnerability detection method and slither showed the good-enough performance. In timestamp dependency vulnerability detection, the proposed CodeNet-based vulnerability detection method showed the best performance, i.e., 94.35% for *accuracy*, 98.66% for *recall* and 94.35% for *f1-score*. Also, Osiris and Oyente showed very low *recall* and Mythril did not detect any *timestamp dependency* vulnerabilities.

In Fig. 7, we show the average detection results of six state-of-the-art tools and the proposed CodeNet-based vulnerability detection method for the smartbug^{wild} dataset. The proposed CodeNet-based vulnerability detection method showed the best performance on *accuracy*, *recall* and *f1-score* by as much as 97.66%, 98.57% and 97.63%, respectively. Even though *precision* of the proposed CodeNet-based vulnerability detection method showed worse results than the others, the proposed CodeNet-based vulnerability detection method showed good-enough result by as much as 96.79%.

In Table 3, we compared the detection performance of the proposed CodeNet-based vulnerability detection method with six state-of-the-art static tools for smartbug^{curated} dataset, where there is no *Tx.origin* vulnerability. Since there exist no labels for benign dataset, we only measured *recall*. In *reentrancy* vulnerability detection, Oyente, Slither, Smartcheck and the proposed CodeNet-based vulnerability detection method showed good performance, which is similar to that for the smartbug^{wild} dataset. In *unchecked low-level calls* vulnerability detection, Mythril, Smartcheck and the proposed CodeNet-based vulnerability detection method showed the best performance. In *timestamp dependency* vulnerability detection, Smartcheck and the proposed CodeNet-based vulnerability detection method showed the best performance by as much as 60.00%. On average, Mythril showed the best performance by as much as 86.20%. However, Mythril did not detect *timestamp dependency*

TABLE 3. Vulnerability detection rate comparison to static analysis tools on smartbug^{curated} dataset.

Tool	Recall		
	Reentrancy	Unchecked LL Calls	Timestamp Dependency
Mythril	72.41%	**100.00%**	N/A
Osiris	62.07%	N/A	40.00%
Oyente	86.21%	N/A	0.00%
Securify	75.86%	95.83%	N/A
Slither	93.10%	91.67%	**60.00%**
Smartcheck	**100.00%**	**100.00%**	20.00%
CodeNet	**96.55%**	**100.00%**	**60.00%**



(a) VGGNet (b) Inception (c) CodeNet

FIGURE 8. ROC curve for differnet CNN architectures.

TABLE 4. Vulnerability detection *accuracy*, *precision*, *recall* and *f1-score* results comparison to static analysis tools on smartbug^{wild}.

Vulnerability	Evaluation Metric	Architecture		
		VGGNet [6]	Inception [28]	CodeNet
Reentrancy	Accuracy	51.14%	97.14%	98.27%
	Precision	N/A	99.61%	97.65%
	Recall	0%	94.51%	98.84%
	F1-score	N/A%	97.00%	98.24%
Unchecked LL Calls	Accuracy	48.64%	62.25%	**98.79%**
	Precision	48.62%	**99.72%**	99.24%
	Recall	**99.81%**	22.41%	98.26%
	F1-score	65.39%	36.60%	**98.75%**
Tx.origin	Accuracy	50.40%	90.31%	**99.21%**
	Precision	N/A	99.32%	**99.87%**
	Recall	0%	81.01%	**98.53%**
	F1-score	N/A	89.24%	**99.20%**
Timestamp Dependency	Accuracy	47.70%	78.73%	**94.35%**
	Precision	47.74%	70.36%	**90.40%**
	Recall	**99.81%**	95.88%	98.66%
	F1-score	64.59%	81.16%	**94.35%**

vulnerability. The proposed CodeNet showed the best performance by as much as 85.51% on average for *reentrancy*, *unchecked LL calls* and *timestamp dependency* vulnerabilities.

To show that the performance of CodeNet is better than other CNN-based image classification architectures, we measured the performance of well-known CNN architectures, i.e.,VGG and Inception, for smartbug^{wild} dataset. To measure the performance using same dataset, we modified 2-dimensional CNN layer parameter from to 1-dimensional one since CodeNet gets 1-dimension image. Table 4 shows the summary of the evaluation result using different CNN architectures for smartbug^{wild}. We observed that CodeNet showed overwhelming performance for four types of vulnerabilities except for *precision* for *reentrancy* and *unchecked low level calls*. Note that VGGNet model did not learn features from the dataset. This is because the last convolution layer of VGGNet is designed as a fully connected layer, which easily loses the local feature information [15], [33]. Compared to VGGNet, Inception and CodeNet do not lose the local feature information because of GAP and GMP before the last layer respectively [15], [33].

Since *precision* and *f1-score* are calculated by dividing false positives whose values in VGGNet are 0%, the detection results are not available for VGGNet. In *reentrancy* vulnerability detection, even though CodeNet showed best performance in *accuracy*, *recall* and *f1-score*, Inception model showed the best performance in *precision* by as much as 99.16%. In *unchecked low-level calls* vulnerability detection, CodeNet showed overwhelming performance except for *precision*. Even though VGGNet showed the best performance in *recall*, it is because VGGNet always predicted into *True* value. Inception showed bad performance in *accuracy*, *recall* and *f1-score* by as much as 62.25%, 22.41% and 36.30%, respectively while showing good performance in *precision*. In *tx.origin* and *timestamp dependency*, CodeNet outperformed other models. VGGNet showed the best performance in *recall* for *timestamp dependency* vulnerability detection because it always predicted *timestamp dependency* vulnerability into *True*.

To show the performance of CodeNet at all classification thresholds, we observed Receiver operating characteristic(ROC) curves for different CNN architectures. As shown in Fig. 8a, VGGNet shows diagonal lines which mean a random guess. In Fig. 8b, Inception generates a line in the upper left corner of coordinate (0,1) for the *reentrancy* vulnerability, which means low false negatives and false positives. Also, lines for *tx.origin* and *time dependency* vulnerabilities are lower than *reentrancy* vulnerability, which means that there exist more false negatives and false positives than *reentrancy* vulnerability. Compared to these two well-know CNN architectures, CodeNet generates upper left lines for *reentrancy*, *tx.origin* and *time dependency* vulnerabilities, which means low false negatives and false positives.

C. VULNERABILITY DETECTION TIME

To compare the vulnerability detection speed, we measured average and total execution time of six state-of-the-art tools and the proposed CodeNet vulnerability detection method. The results are shown in Table 5. Six state-of-the-art tools using static rules tools. In particular, Mythril, i.e., the slowest speed among evaluation tools, took minutes while detecting vulnerabilities on average. Slither, which showed fastest

TABLE 5. Vulnerability detection time comparison to static analysis tools.

Tools	Average Execution Time				
	Reentrancy	Unchecked LL Calls	Tx.origin	Timestamp Dependency	Total
Mythril	0:01:19	0:01:15	0:01:18	0:01:11	11 days, 18:40:29
Osiris	0:00:32	N/A	N/A	0:00:31	2 days 11:36:53
Oyente	0:00:19	N/A	N/A	0:00:18	1 days 08:31:28
Securify	0:02:51	0:02:49	0:02:49	N/A	19 days, 13:00:02
Slither	0:00:02	0:00:02	0:00:02	0:00:02	07:30:24
Smartcheck	0:00:05	0:00:05	0:00:05	0:00:05	18:35:55
CodeNet	**0.13**	**0.13**	**0.13**	**0.15**	**00:31:39**

speed among state-of-the-art tools, took seconds while detecting vulnerabilities on average. Since CodeNet uses a deep learning technique which shows almost constant execution time, CodeNet took 0.13 seconds, 0.13 seconds, 0.13 seconds, 0.15 seconds for *reentrancy*, *unchecked low-level calls*, *tx.origin* and *timestamp dependency*, respectively. That is, it took only hundreds of milliseconds, which is at least 14 times smaller than other tools. Also, since a deep learning technique can be accelerated using a hardware(HW) accelerator such as graphics processing unit(GPU) and field programmable gate array(FPGA), the vulnerability detection speed of CodeNet can be reduced much as accelerator evolves.

V. CONCLUSION

In this paper, we proposed a new smart contract vulnerability detection method using a new CNN architecture, called CodeNet. From evaluation results, we showed that the proposed CodeNet was much faster than state-of-the-art methods. Under various types of vulnerability, we showed that the proposed method shows higher effectiveness in detection performance and faster detection time than state-of-the-art smart contract vulnerability detection tools. Specifically, to show the vulnerability detection performance of CodeNet, we compared the vulnerability detection performance of CodeNet with existing well-known state-of-the-art vulnerability detection tools. On average, CodeNet showed the best performance on accuracy, recall and f1-score with 98.79%, 98.26%, and 98.75% respectively. Also, the CodeNet takes only hundreds of milliseconds, which is at least dozens of times faster compared to the other tools. From these evaluation results, we believe that the proposed method can help to provide more accurate and useful smart contract vulnerability detection to improve the smart contract development environment.

APPENDIX

We overview examples which show four representative smart contract vulnerabilities, i.e., *reentrancy*, *unchecked low level calls*, *timestamp dependency* and *tx.origin*.

A. REENTRANCY

Listing 1 shows SimpleDAO contract. SimpleDAO contract has donate function which donates ether to contracts

Reentrancy

```
1
2  contract SimpleDAO {
3    mapping (address => uint) public credit;
4    function donate(address to) payable {
5      credit[to] += msg.value;
6    }
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        bool res =
              msg.sender.call.value(amount)();
10       credit[msg.sender] -= amount;
11     }
12   }
13   function queryCredit(address to) returns
         (uint){
14     return credit[to];
15   }
16 }
```

Listing 1. SimpleDAO contract.

and withdraw function to withdraw his donated ether from the contract. Withdraw function has reentrancy vulnerability because it uses msg.sender.call.value function to withdraw ether in Line 8 before updating state variable in Line 9. Listing 2 shows attack contract example. Using a fallback function in Line 5 in Listing 2, the attacker reenter SimpleDAO's withdraw function. When withdraw function in Line 6 in Listing 1 is called, SimpleDAO contract checks whether the account has enough money or not in Line 7. Although previous withdraw function is not completed and the contract state is not updated yet in Line 9 in Listing 1, call.value function sends ether to the attacker contract accounts and call the attacker's fallback function in Line 5 in 2 again. It repeats until SimpleDAO contract's ether decreases to zero.

B. UNCHECKED LOW LEVEL CALLS

Listing 3 shows unchecked low level calls example. Unchecked_call contract sends _amounts to msg.sender using send function in Line 8 after decreasing balances and etherLeft variable values in Lines 6-7. If send function fails in Line 8, previous variable values should be restored. But these low level functions do not restore the previous execution. As a result, despite the failure, balances and etherLeft variable values will be decreased.

```
contract ReenAttack {
   SimpleDAO public dao =
       SimpleDAO(0xd914...39138);
   address owner;
   function ReenAttack(){owner =
       msg.sender; }
   function() {
       dao.withdraw(dao.queryCredit(this));
       }
   function getJackpot(){
       owner.send(this.balance); }
}
```

Listing 2. ReenAttack contract.

Unchecked Low Level Calls

```
contract Unchecked_call {
   uint256 etherLeft == 1000;
   mapping (address => uint256) public
       balances;
   function withdraw(uint256 _amount)
       public {
     require(balances[msg.sender] >=
         _amount);
     balances[msg.sender] -= _amount;
     etherLeft -= _amount;
     msg.sender.send(_amount);
   }
}
```

Listing 3. Unchecked_call contract.

C. TIMESTAMP DEPENDENCY

Listing 4 is Lotto contract that randomly select winner by block.timestamp hash. If the miner check block.timestamp hash value before the mining, the miner will win the jackpot.

Timestamp Dependency

```

contract EtherLotto {
   uint constant TICKET_AMOUNT = 10;
   uint constant FEE_AMOUNT = 1;
   address public bank;
   uint public jackpot;
   function EtherLotto() {
      bank = msg.sender;
   }
   function play() payable {
      assert(msg.value == TICKET_AMOUNT);
      jackpot += msg.value;
      var random =
          uint(sha3(block.timestamp)) % 2;
      if (random == 0) {
         bank.transfer(FEE_AMOUNT);
         msg.sender.transfer(jackpot -
             FEE_AMOUNT);
         jackpot = 0;
      }
   }
}
```

Listing 4. EtherLotto contract.

D. TX.ORIGIN

Listing 6 is TxAttackWallet contract to exploit TxUserWallet in Listing 5. If the victim is phished and sends some ether to TxAttackWallet, the attacker can steal the ether from TxUserWallet because the attacker can get the original address in Line 14 in Listing 6. It will call transferTo function in Line 7 in Listing 5 with the owner's address.

Tx.origin

```
1
2  contract TxUserWallet {
3     address owner;
4     function TxUserWallet() public {
5        owner = msg.sender;
6     }
7     function transferTo(address dest, uint
           amount) public {
8        require(tx.origin == owner);
9        dest.transfer(amount);
10    }
11 }
```

Listing 5. TxUserWallet contract.

```
1
2  interface TxUserWallet {
3     function transferTo(address payable
           dest, uint amount) external;
4  }
5
6  contract TxAttackWallet {
7     address payable owner;
8
9     constructor() public {
10       owner = msg.sender;
11    }
12
13    function() external {
14       TxUserWallet(msg.sender).transferTo(
15       owner, msg.sender.balance);
16    }
17 }
```

Listing 6. TxAttackWallet contract.

REFERENCES

[1] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2020, pp. 454–469.

[2] V. Buterin. (2013). *Ethereum White Paper*. [Online]. Available: https://ethereum.org/en/whitepaper/

[3] Y. Chinen, N. Yanai, J. P. Cruz, and S. Okamura, "RA: Hunting for re-entrancy attacks in Ethereum smart contracts via static analysis," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Nov. 2020, pp. 327–336.

[4] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 1251–1258.

[5] CoinDesk. (2016). *Understanding the DAO Attack*. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists

[6] ConsenSys. (2018). *Mythril*. [Online]. Available: https://github.com/ConsenSys/mythril-classic

[7] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, Jun. 2020, pp. 530–541.

[8] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May 2019, pp. 8–15.

[9] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.* New York, NY, USA: ACM, Jul. 2020, pp. 415–427, doi: 10.1145/3395363.3397385.

[10] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Analyzing the out-of-gas world of smart contracts," *Commun. ACM*, vol. 63, no. 10, pp. 87–95, Sep. 2020.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[12] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "EOSAFE: Security analysis of EOSIO smart contracts," in *Proc. 30th USENIX Secur. Symp. (USENIX Secur.)*, 2021, pp. 1271–1288.

[13] T. H.-D. Huang and H.-Y. Kao, "R2-D2: ColoR-inspired convolutional NeuRal network (CNN)-based AndroiD malware detections," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 2633–2642.

[14] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.* New York, NY, USA: ACM, Sep. 2018, pp. 259–269.

[15] M. Lin, Q. Chen, and S. Yan, "Network in network," 2013, *arXiv:1312.4400*.

[16] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "ReGuard: Finding reentrancy bugs in smart contracts," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng., Companion*, May 2018, pp. 65–68.

[17] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.

[18] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2019, pp. 1186–1189.

[19] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 653–663.

[20] S. Repository. (2021). *Solidity*. [Online]. Available: https://github.com/ethereum/solidity/

[21] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 757–762.

[22] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Represent.*, 2015, pp. 1–14.

[23] SmartBugs. (2020). *Smartbugs Wild Dataset*. [Online]. Available: https://github.com/smartbugs/smartbugs-wild

[24] SmartBugs. (2021). *SB Curated: A Curated Dataset of Vulnerable Solidity Smart Contracts*. [Online]. Available: https://Github.com/smartbugs/smartbugs/tree/master/dataset

[25] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VERISMART: A highly precise safety verifier for Ethereum smart contracts," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1678–1694.

[26] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2818–2826.

[27] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, May 2018, pp. 9–16.

[28] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in Ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 664–676.

[29] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 67–82.

[30] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "ContractWard: Automated vulnerability detection models for Ethereum smart contracts," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 1133–1144, Apr. 2021.

[31] M. Wohrer and U. Zdun, "Smart contracts: Security patterns in the Ethereum ecosystem and solidity," in *Proc. Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Mar. 2018, pp. 2–8.

[32] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Apr. 2014. [Online]. Available: https://github.com/ethereum/yellowpaper

[33] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2921–2929.

SEON-JIN HWANG received the B.E. and M.S. degrees from Pusan National University, Busan, South Korea, in 2019 and 2021, respectively, where he is currently pursuing the Ph.D. degree in computer science and engineering. His research interests include security for artificial intelligence, blockchain, and intrusion detection.

SEOK-HWAN CHOI received the B.E. degree from Pusan National University, Busan, South Korea, in 2016, where he is currently pursuing the Ph.D. degree in computer science and engineering. His research interests include security for artificial intelligence, adversarial examples, and intrusion detection.

JINMYEONG SHIN received the B.E. degree from Pusan National University, Busan, South Korea, in 2017, where he is currently pursuing the Ph.D. degree in computer science and engineering. His research interests include security for artificial intelligence, homomorphic encryption, and intrusion detection.

YOON-HO CHOI (Member, IEEE) received the M.S. and Ph.D. degrees from Seoul National University, Seoul, South Korea, in 2004 and 2008, respectively. From September 2008 to December 2008, he was a Postdoctoral Scholar with Seoul National University. From January 2009 to December 2009, he was a Postdoctoral Scholar with Pennsylvania State University, University Park, PA, USA. While working as a Senior Engineer with Samsung Electronics, from May 2010 to February 2012, he had been deeply involved in the development of a commercial long-term evolution cloud communication system. He was an Assistant Professor at Kyonggi University, Suwon, South Korea, from May 2012 to August 2014. He is currently a Professor with the School of Computer Science and Engineering, Pusan National University, Busan, South Korea. His research interests include AI security, privacy preserving machine learning algorithms, adversarial deep learning, blockchain, deep packet inspection, and anomaly detection algorithms. He has served as a member for several technical program committees in various international conferences and journals.