Making Smart Contract Development More Secure and Easier

Meng Ren
Tsinghua University
Beijing, China

Fuchen Ma*
Tsinghua University
Beijing, China

Zijing Yin
Tsinghua University
Beijing, China

Ying Fu
Ant Financial
Beijing, China

Huizhong Li
WeBank
Shenzhen, China

Wanli Chang
University of York
York, UK

Yu Jiang†
Tsinghua University
Beijing, China
jiangyu198964@126.com

ABSTRACT

With the rapid development of distributed applications, smart contracts have attracted more and more developers' attentions. However, developers or domain experts have different levels of familiarity with specific programming languages, like Solidity, and those vulnerabilities hidden in the code would be exploited and result in huge property losses. Existing auxiliary tools lack security considerations. Most of them only provide word completion based on fuzzy search and detection services for limited types of vulnerabilities, which results in the manpower waste during coding and potential vulnerability threats after deployment.

In this work, we propose an integrated framework to enhance security in the two stages of recommendation and validation, assisting developers to implement more secure contracts more quickly. First, we reinforce original smart contracts with general patch patterns and secure programming standards for training, and design a real-time code suggestion algorithm to predict secure words for selection. Then, we integrate multiple widely-used testing tools to provide validation services. For evaluation, we collected 47,398 real-world contracts, and the result shows that it outperforms existing platforms and tools, improving the average word suggestion accuracy by 30%-60% and helping detect about 25%-61% more vulnerabilities. In most cases, our framework can correctly predict next words with the probability up to 82%-97% within top ten candidates. Compared with professional vulnerability mining tools, it can find more vulnerabilities and provide targeted modification suggestions without frivolous configurations. Currently, this framework has been used as the official development tool of WeBank and integrated as the recommended platform by FISCO-BCOS community.

*Fuchen Ma have contributed equally to this work.
†Yu Jiang is the corresponding author.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3473929

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Smart Contract Development; Domain-specific Reinforcement; Integrated Testing

ACM Reference Format:
Meng Ren, Fuchen Ma, Zijing Yin, Ying Fu, Huizhong Li, Wanli Chang, and Yu Jiang. 2021. Making Smart Contract Development More Secure and Easier. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3473929

1 INTRODUCTION

With the increase of application scenarios, smart contracts are attracting more and more users [2, 5, 17, 36]. Different from traditional applications implemented by programming languages such as C and Java, smart contracts written in Solidity often have complex domain-specific business logic and numerous distinctive features, such as the gas system. Due to the distributed execution environment, the complexity and limitations of programming languages, developing secure smart contracts can be challenging for most developers, even the experienced ones [10]. For convenience, lots of assistance tools have been developed. Editors such as Remix [13], solidity-plugin [22] and YAKINDU [46] are applied for contract coding, which supports highlighting, compiling, and debugging. Professional analyzers such as SmartCheck [40] and ContractFuzzer [21] check the code logic based on static analysis or dynamic testing to avoid vulnerability attacks after deployment.

Those existing works have done well to help engineers in smart contract development, but there are still many limitations for real industrial practice. First, for most existing coding platforms such as Remix, they only support auto-completion based on fuzzy match, which relies on static analysis of libraries and referred files to provide suggestions about tokens such as keyword names or API list. They are not able to deal with the contextual relevance and data dependency between current location and previous text, therefore fail to capture deeper semantic association or provide more targeted

suggestions with security reinforcement. Second, those existing platforms usually integrated little or no validation tools, and need extra configurations of different analyzers for vulnerability detection. Furthermore, since those validation tools are also independent with each other, there would be high number of false negatives when running separately. Due to the immutability of block-chain data, the owner cannot apply any patch to the original contract and the hidden vulnerabilities may be exploited and lead to unpredictable loss. Though the author can remedy it by deploying a new contract, this process is far more complicated and may yield higher transaction costs. Therefore, it is great urgent to develop a unified platform to help engineers during coding and testing.

To further improve the programming efficiency and ensure the security of code, we need to solve two challenges:

- **How to recommend not only legal but also secure words for selection?** First, we need to build a language model to capture the semantic dependencies between contexts. Existing Solidity editors use either static analysis or word matching to suggest tokens such as keyword names or API, they can not handle semantic association and are short of accuracy. Although traditional C or Java code completion techniques apply language model and deep learning to predict more suitable tokens, how to transfer them into programming language of Solidity still needs meticulous design. Furthermore, the completion should also ensure security. Unlike the traditional code completion technology that mainly focuses on the correctness, Solidity code needs to pay extraordinary attention to security. Since a contract cannot be revised once deployed, the crawled contracts for training may violate secure programming standards and contain vulnerable operations. Therefore, we need to define a series of security patches to reinforce them and try to learn and recommend those safe operations, such as APIs in SafeMath libraries [31].
- **How to conduct a comprehensive security analysis of smart contracts effectively and quickly?** Currently, there are lots of tools used for smart contract testing, but each tool only supports the detection of limited types of vulnerabilities. Furthermore, most of them are separated with the coding editors, and need frivolous configurations for execution. If the user wants to perform a more comprehensive inspection of the contract, he must use different interfaces and sort out the inspection results by himself, which is very inefficient and user-unfriendly.

To address the above challenges, we propose an integrated framework, which provides safety-related assistance throughout the development process. It mainly consists of two components, security-reinforced code suggestion and security-oriented code validation. For security-oriented code suggestion, we first perform security reinforcement on original contracts crawled from block-chain network. Then, we locate and fix pre-defined security issues based on AST and Datalog. After preparation, we build a language model and a context-based word-selection algorithm to provide reasonable words for users. For security-oriented validation, in order to make up for the limitation of separate analysis tools and fuse them with the implementation smoothly, we integrate five free and open-source vulnerability detection tools and organize their output from a global perspective.

For evaluation, we implemented the user interaction logic based on VS Code, and collected 47,398 real-world deployed smart contracts for training and verifying. The results show that our framework outperforms existing auxiliary tools. For example, compared with Remix, it improves the average word suggestion accuracy by 30%-60%, and detects about 25% more vulnerabilities. Furthermore, it can correctly predict the next security-reinforced token with the probability up to 82%-97% within top ten candidates and 74%-91% within top five candidates. Compared with other professional bug finders, our framework is able to detect more vulnerabilities, with about 54 types, and provide targeted modification suggestions, while other tools only support 10-20 types when running alone.

Contributions We mainly make the following contributions:

- We designed a security-reinforced code suggestion module based on bidirectional LSTM network, and conducted contract reinforcement before training for potential risks such as integer overflow and out-of-gas error, based on AST and Datalog.
- We built a security-oriented code validation module based on five widely-used analyzers to detect vulnerabilities and backdoor threats hidden in smart contracts. It can not only support 54 types of security problems, but provide detailed description and revise suggestions as well.
- The framework has been used as the official development tool of WeBank and integrated as the recommended platform by FISCO-BCOS community[1], which is a famous financial block-chain cooperation alliance.

2 BACKGROUND ON SMART CONTRACT

```
pragma solidity >=0.4.22;

contract Storage {
    uint storeData;
    function addData(uint x) returns (uint) {
        return storeData + x;
    }
}

contract Owned {
    constructor() public { owner = msg.sender; }
    address owner;
    function setOwner(address newer) public {
        require(msg.sender == owner);
        owner = newer;
    }
}

contract BigStore is Storage, Owned {
    uint nowStore;
    function get() view public {
        return storeData;
    }
    function bigStore(uint a, uint b) public {
        if (a > b) { nowStore = addData(a); }
        else { nowStore = addData(b); }
    }
}
```

Listing 1: The source code of *BigStore* contract.

[1] https://github.com/FISCO-BCOS/SCStudio

In block-chain system, smart contracts are tiny programs deployed on the chain. They are a set of promises defined in digital form, which control the digital assets and contain the rights and obligations agreed by the contract participants. They are immutable and run in a distributed way, automatically execute the code logic and maintain tEffectivenesshe credibility.

Instead of directly writing bytecode, developers will choose other high level languages for development. There are many programming languages for writing smart contracts: Serpent, Solidity, Viper, and so on. The most widely used one is Solidity [44] developed by Ethereum project's Solidity team, which is an ECMAScript-like language with enhanced static types, contract-related features, and many other functionalities. The Solidity code will be compiled into bytecode and run on the Ethereum Virtual Machine, which is a stack machine that natively supports an Ethereum-specific bytecode with an instruction set. Each contract will be allocated with a unique address. After deploying onto the block-chain, the user can run a specific function by invoking a transaction to the associated address of the contract.

Listing 1 is an example of smart contracts implemented by Solidity. Contract *Storage* is used to record the value of stored data and provides some modification methods, such as adding. Contract *Owned* stores the owner, defines a *setOwner* method to modify the owner. *BigStore* inherits these two contracts and realizes further functions, like storing the bigger object.

3 METHODOLOGY

In this section, we introduce the overall workflow of our framework and each component in detail. As shown in Fig. 1, the front-end is mainly responsible for monitoring the user status and displaying the information, and the back-end is responsible for code suggestion and validation. While the user is coding, it will monitor the current status in real time. When it receives a "Space", all the previous text will be packaged and sent to the back-end through a POST request. Then the code suggestion module will call the language model to generate a list of possible next-words, then package them into a JSON file to return. When the user has finished coding or imported existing smart contracts, he can perform validation by invoking the security analysis command. At this time, the front-end will package all the codes in current editing interface and send them to the server. After executing the built-in detection tools in parallel and integrating their output information, a detailed bug report will be generated and sent back to the front-end.

3.1 Security-Reinforced Code Suggestion

Smart contracts have strict requirements on the logic of the code. Reasonable code recommendations with security constraints can effectively prevent embedded vulnerabilities, thereby reducing the workload of code review and modification. In order to ensure the safety of the recommended code, we must first build a dataset that contains hardened contracts without security risks, and then feed them into the language model. First, we need to strengthen the crawled contracts based on general patch patterns for vulnerabilities and secure programming standards, such as the use of SafeMath library. Then, we performed symbol substitution on user-defined information, such as variable names, function names and contract names. Next, we used these data to train a bidirectional LSTM network with attention mechanism. Finally, we performed a context-based word selection algorithm to replace these special symbols with the possible valid words. Details are as below.

3.1.1 Contract Reinforcement. Unlike the traditional C and Java code that we can collect the correct code for training, smart contracts deployed on the chain cannot be revised due to the immutability. According to [12, 30, 34], 97% of deployed contracts on Etherscan [14] is vulnerable. Hence, we should do the reinforcement to add security constraints and general patches to the original contracts before model training for a secure prediction.

Compared with modifying the source code directly, tree structure is more convenient for addition, deletion and modification of modules, such as replacing the binary operation with a safe function call. As a supplement, Datalog is suitable for checking internal logic and can quickly locate corresponding statements. Therefore, we performed reinforcement based on Abstract Syntax Tree (AST) and Datalog in turn.

Based on AST, we developed an automatic alteration tool to strengthen the original Solidity codes with unified standards. It takes AST information and text formats generated by solc compiler [39] as an input, and traverses all the AST nodes to check the type, then apply different modification to each type of nodes. Taking the reinforcement of integer overflow threat as an example, we will replace all arithmetic operations with safe functions encapsulated in the SafeMath library and use the modified AST to generate a new piece of code, as shown in Table 1.

Table 1: Examples of reinforcement based on AST

Original Solidity Code	After Reinforcement
... a + b ...	... a.add(b) ...
... a - b ...	... a.sub(b) ...
... a * b ...	... a.mul(b) ...
... a / b ...	... a.div(b) ...

After the reinforcement of secure programming standards, we are able to further strengthen the original Solidity codes with general patch patterns for other existing vulnerabilities such as out-of-gas errors. We use Datalog to define rules and related data structures for the semantic facts extraction, based on which, we can perform pattern matching to find vulnerable structures or calling sequences, and accomplish the reinforcement through modifying statements in static single assignment form. Take the patch to out-of-gas error as an example. As defined in MadMax [15], there are three common patterns of gas-focused vulnerabilities:

(1) Unbounded Mass Operations: Loops whose behavior is determined by user input could iterate too many times, becoming too economically expensive.
(2) Non-Isolated External Calls (Wallet Griefing): Using *send* without a check of the result in a loop, which will execute the callback function of caller and the attacker can provide a callback function that runs out of gas.

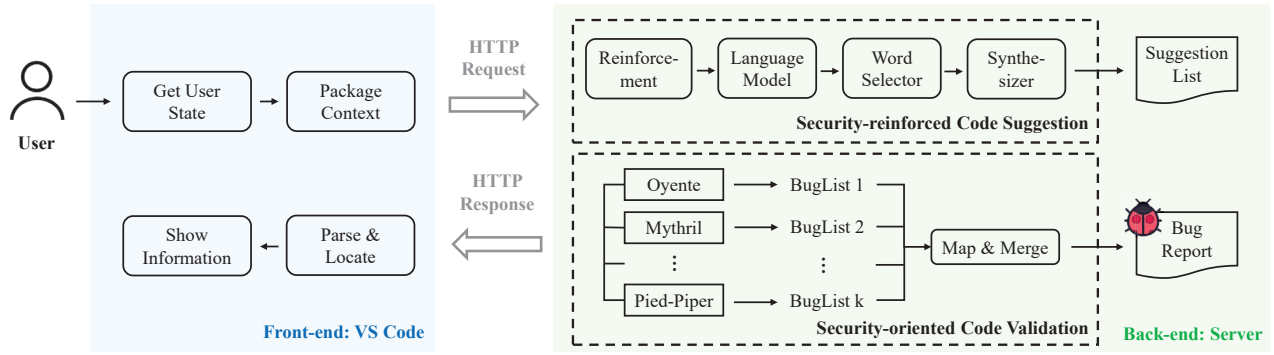

Figure 1: The flowchart of our framework. The front-end will monitor the editing status of users in real time, package the interface information to send a request, and parse the return file. The back-end will process the request, and call security-oriented code validation module for testing or security-reinforced code suggestion module for recommendation.

(3) Integer Overflows: When using *var* to declare the iteration variable, a mere addition beyond the range of *uint8* can make the loop unable to terminate.

Table 2: Examples of reinforcement based on Datalog

Original Solidity Code	After Reinforcement
for (uint i = 0; i <accounts.length; i++) { ... }	for (uint i = 0; i <accounts.length; i++) { if (msg.gas <100000) break; ... }
for (...) { ... investors[i].addr.send(val) ... }	for (...) { ... investors[i].addr.transfer(val) ... }
for (var i = 0; i <condition; i++) { ... }	for (uint i = 0; i <condition; i++) { ... }

According to these definitions, we can define a for-loop structure in Datalog and further check whether it contains *var* in the header, or internally involves a *send* operation that does not check the returned result, etc. If yes, reinforcements are conducted. The original solidity code and reinforced code are presented in Table 2.

We use the reinforcement of SafeMath library to avoid overflow threats and pattern based patch to avoid out-of-gas error for demonstration, and other common smart contract issues can be reinforced with similar operations. Training on these reinforced contracts will enhance the security of recommended code.

3.1.2 Symbol Substitution. For the language model, some words are of little learning value and may even have a negative impact on training, such as the name of user-defined variables, functions, parameters and contracts. A more reasonable process is to predict a framework first, and then fill the frame based on user-defined information in current context.

To quickly locate the position of different variables, we traverse the child nodes of each statement based on the AST, and then determine whether the type of each node belongs to a variable name, a function name, a contract name or other type. For the first three cases, if the node is a built-in type, it will not be modified. Otherwise, it will be replaced by $VARIABLE$, $FUNCTION$ or $CONTRACT$ automatically.

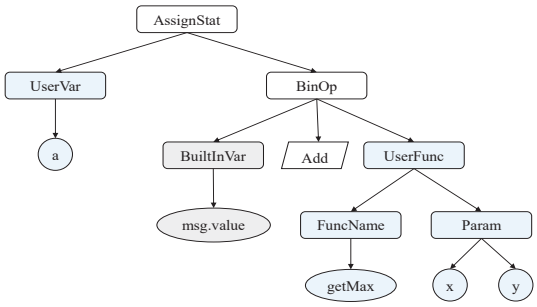

Figure 2: The AST of the sample code. The shaded part in gray is the subtree where the built-in type is located, and the shaded part in blue is the subtree where the user-defined content is located, which should be substituted.

Take $uint\ a = msg.value + getMax(x, y)$ as an example, Fig. 2 shows the corresponding abstract syntax tree. Since *msg.value* is an embedded variable, a is a user-defined variable, *getMax* is a user-defined function and x, y are its receiving parameters, a, x, and y will be replaced by $VARIABLE$ and *getMax* will be replaced by $FUNCTION$, while *msg.value* remains unchanged.

3.1.3 Making Predictions. After building the training dataset, we started to train our language model. Each sentence will be divided into a series of word segmentation to form question-answer pairs, making preparation for feeding into the deep learning network. We used a tokenizer to realize the sub-word division and filter the low-frequency words, such as some fixed addresses related to a specific project.

In order to capture the context dependencies in a long distance, we choose LSTM network as the basic language model. However, sometimes the prediction of current token may need to consider both the previous inputs and the subsequent inputs, which will

Table 3: Summary of selected tools and their characteristics

Tool Name	Environment	Analysis Level	Methods	Description
Oyente [27]	Python	source code and byte code	Symbolic Execution	Oyente is a symbolic execution tool exclusively designed to analyze Ethereum smart contracts.
Mythril [9]	Python	source code and byte code	Concolic Analysis	Mythril is a security analysis tool for Ethereum smart contracts, which can detect a range of security issues.
Securify [41]	Java	source code and byte code	Datalog Analysis	Securify is a scalable analyzer for Ethereum smart contracts, and is able to prove contract behaviors as safe/unsafe with respect to a given property.
SmartCheck [40]	Java	source code	Pattern Matching	SmartCheck is an extensible static analysis tool that checks XML-based intermediate representation of Solidity source code against XPath patterns.
Pied-Piper [43]	Java	source code and byte code	Datalog Analysis	Pied-Piper is a static analysis tool that constructs CFG based on bytecode and extracts semantic facts to detect potential backdoors hidden in smart contracts.

make the result more accurate. Therefore, on top of the forward network, we built another layer of reverse network. In the forward layer, the calculation is performed from T_1 to T_t, then the output of the forward hidden layer at each time is obtained and saved. In the backward layer, the calculation is performed in the reverse direction to obtain the output of each hidden layer. At last, the final output is obtained by combining the results of the forward and backward layers. This process is presented in equations 1 to 3.

$$s_t = f(Ux_t + Ws_{t-1}) \tag{1}$$

$$s'_t = f(U'x_t + W's'_{t+1}) \tag{2}$$

$$h_t = g(Vs_t + V's'_t) \tag{3}$$

where x_t is the input word, s_t is the hidden state of each LSTM cell in the forward direction, s'_t represents the corresponding state in the reverse direction, h_t is the hidden state output of bidirectional LSTM network at time t, U, W, V, U', W', V' are trainable parameters and f, g are activation functions.

Inspired by [26, 47], we further introduced the attention mechanism to capture the most important semantic information within all context. Firstly, we calculated the attention weight matrix α_t and built a global context vector c_t by weighted averaging all the hidden states $[h_1, h_2, ..., h_t]$. Subsequently, we obtained the output vector o_t by concatenating c_t and current hidden state h_t, which encodes the next-word distribution and is projected to the size of the vocabulary. Finally, we applied a $softmax$ layer to get the final probability distribution y_t. All steps are performed according to equations 4 to 8, where W_a, W_c, W_s are trainable parameters. After training, the language model can predict top-k candidate words according to the last sentence and scenario context.

$$H = [h_1, h_2, ..., h_t] \tag{4}$$

$$\alpha_t = softmax(W_a h_t) \tag{5}$$

$$c_t = H\alpha_t^T \tag{6}$$

$$o_t = tanh(W_c[c_t; h_t]) \tag{7}$$

$$y_t = softmax(W_s o_t) \tag{8}$$

3.1.4 Symbol Recovery. Sometimes, the word list provided by the language model is not complete, it may contain some substituted symbols, such as $VARIABLE$, $FUNCTION$ and $CONTRACT$. So we need to design a word selection algorithm to fill these vacant parts.

There are two forms of a vacant word, one is a newly defined object, and the other is a reference object that has appeared in the context. To meet the first requirement, we need to build a default database. Since the data we used are real contracts crawled from the engineering projects, we directly extracted the variable names, function names and contract names from them, and recorded the number of their occurrences in the entire dataset. For the second case, a father tree structure will be constructed based on the cursor position and accessible relations in the context. Then, the defined variables, functions and contracts of all the father nodes will form a user-defined information library, and the relevant distance information will also be recorded.

After the preparation, word selection algorithm starts. The first step is to judge the character of the vacant word based on sentence pattern. If the symbol to be replaced is in a declaration statement and is a declared object, the candidate word will be selected from the default dataset based on the occurrence frequency. Otherwise, the candidate word will be selected from the user-defined information library based on semantic distance.

3.2 Security-Oriented Code Validation

When the programming work is completed, the code validation module will conduct a comprehensive testing of current contract against common security risks. Though there has been many efforts from the research community to develop automated analysis tools, each of them supports different range of vulnerability types and depends on different environment configurations. If developers want to scan the contract comprehensively, they need to run each analysis tool separately and screen its output manually. During this process, they will face the following problems: (1) Each tool has different naming and locating methods for the same kind of vulnerability. Users need to understand the meaning of each definition first, and then judge the repeated warnings in each report file

Table 4: Categories of high-risk smart contract vulnerabilities and the level where the attack can be mitigated.

Category	Description	Level
Access Control	Failure to use function modifiers or use of tx.origin.	Solidity
Arithmetic	Integer overflow/underflow.	Solidity
Backdoor Threats	High-privilege functions, which can only be invoked by certain group of accounts.	Solidity
Front Running	Two dependent transactions that invoke the same contract are included in one block.	Blockchain
Locked Ether	The withdraw function of a transferable contract becomes unable to send tokens.	Solidity
Reentrancy	Reentrant function calls to another untrusted contract before resolving any effects.	Solidity
Timestamp Dependency	The timestamp of the block is manipulated by the miner.	Blockchain
Unchecked Low Calls	call(), callcode() or delegatecall() fails and it is not checked.	Solidity
Unhandled Exception	Uncaught exceptions.	Solidity

through independent analysis; (2) When one tool reports an error but another tool does not, developers can not believe the testing result of any of them, so they need further manual analysis or seek the help of more detection tools. Therefore, integrated security testing can greatly speed up the development efficiency and reduce unnecessary workload of developers.

With an empirical analysis of public smart contract testing tools, we decided to integrate five free and open-source tools to build a comprehensive testing platform. They are implemented based on different technologies and meet the requirements of covering all common security issues of smart contracts with the smallest number of integrations. Table 3 shows the summary of the selected tools and their main properties. The selected tools are based on different detection methods and guarantee the security of contract in different aspects. Combining with static and dynamic analysis, they can cover the most types of smart contract vulnerabilities. They also meet the requirement of open-source, available command line interface and supporting source code level analysis and location.

Oyente [27] is based upon symbolic execution, which uses symbolic value instead of specific variables to simulate each path to generate the possibility of each execution. It first decompiles the bytecode to construct the control flow graph, and then symbolically executes the contract based on the current Ethereum global state to produce a set of symbolic traces. Finally, it uses the pre-defined logic to check whether there exists a vulnerability.

Mythril [9] uses static analysis, taint analysis and concolic execution to discover real-world issues. It draws the control flow graph first, and simulates the execution of each path with symbols. It also uses the control flow to prove certain properties of contracts, determining reachability of error states.

Securify [41] analyzes each module of contract after decoupling. It first converts contract bytecode to Datalog and extracts semantic facts. Then it transforms the vulnerabilities into a series of compliance and violation patterns and defines related data structures. Finally, it traverses the facts to search for unsafe patterns.

SmartCheck [40] also constructs an internal representation to check the contract. It runs lexical and syntactic analysis on reliable source code. It uses ANTLR [32] and a set of Solidity syntax to generate an XML parsing tree, then detects vulnerability patterns by using XPath queries on IR.

Pied-Piper [43], which aims to detect a set of high-privilege functions named backdoors instead of common security issues, first collects semantic facts through control flow graph and then builds high-level data structures and relations. For each type of backdoor, Pied-Piper decomposes it into a series of semantic units and uses logical operation like "AND" or "OR" to connect them. After that, it performs a domain-specific Datalog analysis based on a Datalog engine named Vandal [7] to identify each threat.

In the process of integration, we first configured the environment that each tool depends on and deployed it on the server. Then, with the help of command-line based call interface, we implemented a script to call each tool in parallel. When all tools have finished execution or reached the preset time, we extracted the vulnerability type, location and other information from the output file to form the final report. According to the documents of each tool, we summarized the characteristics of each vulnerability and mapped them to the objects defined in our vulnerability library with unique IDs. For the same object reported by different tools on the same line, we considered it as a duplicate warning. To further reduce false positives, we set a judgment threshold k for the vulnerabilities that are supported by more than one tool. That is, only when there are more than k tools confirm the existence of this vulnerability, we will consider it as a real warning. In practice, k is set to $\lceil N/2 \rceil$, where N is the number of tools that can detect this kind of bug. With unified execution, it frees the engineers from frivolous configurations and improves the efficiency.

4 IMPLEMENTATION

Model Training. We first utilized SIF [33] to perform symbol substitution on the training contracts, and then divided each sentence into a series of tokens. Using training settings consistent with traditional language models, we selected 20 consecutive tokens as the "question", and the following token as the "answer". After making the question-answer pairs, we respectively built two dictionaries to store unique tokens in the question and answer part. According to the dictionary index, we converted each question sequence into an integer vector with a maximum length of 20, and encoded the answer token into a one-hot vector with a length of 14614. For further compression, we added an embedding layer in front of the network to convert the integer vector into a low-dimension vector with a length of 300, then fed them to the LSTM layer.

The neural language model was developed in Tensorflow [1] and Keras [8], and trained using cross-entropy loss and mini-batch SGD with a batch size of 2048. Since the memory space could not

Table 5: Accuracy of security-reinforced code suggestion. Remix only supports token level suggestion with the first prefix character, while our framework accomplishes the suggestion without the need of any prefix character information.

Project	#Contracts	LoC	#Tokens	Top-1		Top-3		Top-5		Top-10	
				Remix	Ours	Remix	Ours	Remix	Ours	Remix	Ours
Airdrop	4	96	745	9.52%	63.64%	17.32%	73.24%	42.86%	77.62%	57.14%	86.52%
ARIYAX	2	138	812	14.29%	77.27%	14.29%	83.37%	28.57%	88.64%	38.10%	93.77%
CrystalDeposit	14	692	3084	4.76%	70.68%	33.33%	75.41%	33.33%	79.50%	33.33%	85.88%
EthVentures4	8	394	3419	9.09%	53.99%	15.60%	72.19%	22.73%	74.44%	40.91%	81.96%
Eximchain	22	1016	5217	13.64%	69.57%	45.45%	81.28%	45.45%	84.01%	54.55%	91.87%
GasManager	10	66	237	21.74%	73.97%	41.27%	79.34%	43.48%	82.19%	60.87%	89.56%
Ipsx	16	738	4022	8.70%	78.15%	10.36%	84.53%	34.78%	88.74%	52.17%	94.92%
KyberNetwork	8	342	1818	19.05%	80.87%	19.05%	87.97%	23.81%	91.28%	42.86%	96.62%
OneEight	6	286	1405	22.73%	75.98%	25.00%	81.73%	36.36%	84.08%	50.00%	93.11%
UpgradeProxy	18	416	1903	18.18%	68.46%	50.00%	74.21%	50.00%	82.08%	63.64%	88.95%
Average				14.17%	71.26%	27.17%	79.34%	36.14%	83.26%	49.36%	90.32%

support loading such a big dataset at one time, we divided the data into 9529 groups and loaded them in turn. The initial learning rate is set to 0.01 and will decay by half after several epochs. All models use a hidden size of 128, and gradient norm clipping of 15. For attention layer, the size of fixed-window attention memory is set to 20 tokens. As regularizer, we also used a dropout of 0.2 on the linear transformation of the inputs.

Unified Execution. Each tool depends on different environments and software versions. In order to avoid conflicts, we chose to run each tool in Docker [11] container and save the output information locally. After collecting and analyzing the vulnerability types supported by each tool, we built our own library and constructed a mapping between the vulnerability name defined in each tool and the index number in our library. Before calling the tools, we defined a global vulnerability list, which represents the security issues hidden in the tested contract. Then, we created 5 threads to call different detection tools concurrently. After execution, we extracted and merged the vulnerability types from their output files, and filled the corresponding index number into the global list. After completing the name, description, location (corresponding line number), warning level and modification suggestion of each vulnerability, the final report is generated.

Taking full advantage of each tool, our framework can detect 54 common security vulnerabilities, including Top-9 categories of high-risk security issues, such as reentrancy, integer overflow, unhandled exception, and 5 kinds of backdoor threats. Among all supported vulnerabilities, 19 of them are included in the SWC registry [29], which reaches a coverage rate of 52.8%.

Table 6: Smart contracts corpus statistics.

	Train	Dev	Test	Total
#Files	28438	9480	9570	47398
#Lines	3230919	1239021	1223517	5693457
#Unique tokens	93985	38265	35076	157760
#tokens per line	18.26	18.32	17.21	18.07

5 EVALUATION

5.1 Dataset and Environment Setup

All experiments were performed atop a machine with 8 cores (Intel i7-7700HQ @3.6GHz), 24GB of memory, and Ubuntu 16.04.6 as the host operating system. We built a corpus by collecting 47,398 unique real-world smart contracts crawled from Ethereum network [14]. For model training, we divided the large dataset into three parts: training set, verification set and test set. The division ratio is 6:2:2. After compiling and tokenizing, the total size of vocabulary is 93,985 and the number of question-answer pairs is 19,517,495. Table 6 shows the corpus statistics.

In order to test the effectiveness of code validation module, we built a test contract set of size 181 with clear annotations, which consists of 131 vulnerable contracts with 176 tagged vulnerability labels collected by SmartBugs [37] and 50 vulnerable contracts with backdoor labels collected by Pied-Piper [43]. All vulnerabilities are divided into nine different categories (see Table 4), and we ran each tool in turn on each category to make statistics of bug detection.

5.2 Necessity of Oriented Reinforcement

Most developers are unable to ensure that the code is free-of-bug, especially for projects that have been uploaded earlier. As reported in [12], 79% of contracts are flagged as having an arithmetic vulnerability, which means that, in our original dataset, most contracts also have problems related to arithmetic operation. In order to improve code quality, we should reinforce the training dataset first to help our language model recommend as many code pieces with security constraints as possible.

We trained the language model based on the original dataset and the reinforced dataset respectively until the loss converged. Then, following the same process as in the previous paragraph, we counted the occurrences that the two models recommended SafeMath functions instead of vulnerable arithmetic operations at each test point. As shown in Table 7, the model trained with reinforced contracts was able to recommend security functions more frequently, which reveals that the benefit of using security-aware training sets is significant.

Table 7: Frequency statistics of recommended SafeMath functions using models trained with different datasets.

Project	Ori-Dataset	Re-Dataset	Improved By
Airdrop	33	48	45.45%
ARIYAX	35	58	65.71%
CrystalDeposit	151	274	81.46%
EthVentures4	67	163	143.28%
Eximchain	120	185	54.17%
GasManager	14	22	57.14%
Ipsx	82	168	104.88%
KyberNetwork	49	70	42.86%
OneEight	71	97	36.62%
UpgradeProxy	35	71	102.86%

Table 8: Detection of integer over/underflow vulnerabilities.

	Ori-Dataset	Re-Dataset
# Contracts	10191	259
Reported Bugs	64115	2176
True Positive Samples	61939	0

We also used Oyente [27], which is a professional analyzer based on dynamic symbol execution and has a high detection accuracy, to demonstrate the integer security of reinforced contracts. Take integer vulnerabilities as an example, Table 8 shows the results on each dataset. For the original contract set, it labeled 10,191 contracts as vulnerable. Since Oyente has difficulty in distinguishing dynamic array and binary operation, for example, it will mistakenly marks "string public user_name = 'Dola'" as an integer underflow problem. Therefore, we set a review principle to filter out statements that do not contain arithmetic operations. After filtering, there are 61,939 bugs left. We randomly selected 100 of them for checking, and found all of them are true positive samples. For reinforced contract set, only 259 contracts are suspected to contain arithmetic problems, and after manual checking, all of them are false positive samples. In other words, the reinforced contracts are free of integer bugs.

5.3 Accuracy of Code Suggestion

To verify the performance on code suggestion, we built a small corpus by randomly selecting 10 real-world Ethereum projects (108 contracts) and carried out experiments compared with Remix. For each file in project, we called the framework to recommend the next word in turn using the previous context with a length of 20, and calculated the average accuracy. Since Remix can only generate the candidate list after the first prefix character is given, we additionally provided the first letter of the next word for it at each test point, then counted the hits in the candidate list.

As shown in Table 5, the first four columns respectively represent the name of each project, the number of contracts, and total lines of code and tokens contained in them. *Top-K* means the prediction matches the expected answer in K candidates. Consider Top-1 accuracy, in about 71.26% cases, users can find their expected next-word on the top of our suggestion list, that is 5X higher than Remix's. For Top-5 accuracy, the performance can be improved by 47.12% and achieves an average accuracy of 83.26%. When we expand the number of candidates to 10, the accuracy will rise up to 81.96%-96.62%, which is 41% higher than Remix.

There are two key reasons for Remix's low accuracy even under the circumstances when the first character of the token to be completed is given. First, Remix searches context information and the Solidity keyword library to get the recommended list, so it can only recommend the words contained in the library or have appeared in the context. For the newly defined variables, its accuracy is as low as zero. Second, the ranking of candidate words is completely based on the distance. Without the capturing of long-term dependence in context, it has low accuracy for earlier defined variables.

5.4 Effectiveness of Integrated Validation

For code validation, we compared our framework with other contract auditing tools and Remix, whose core is Mythril. There are nine categories of vulnerabilities annotated in the dataset. When a tool reported a vulnerability, we will verify its category and location manually. If both the category and specific line are consistent with the annotation, we will mark that the tool has successfully detected this vulnerability.

Table 9 summarizes the results of each tool. The first column represents the category of vulnerabilities, the next two columns, *Files* and *Vulns*, represent the number of contracts and bug annotations contained in each category respectively, and the subsequent columns present the number of vulnerabilities detected by each tool in each category. When developers only use one tool, at most 56.6% of hidden vulnerabilities can be detected. In order to avoid huge losses in the future, they have to constantly use different tools for validation, which not only causes a waste of time, but also brings difficulties for developers to eliminate repeated alarms with similar meanings reported by different tools. With integrated testing, we can cover all categories of vulnerabilities in the dataset, and successfully exposed 81.9% of hidden security issues. That is, developers do not need to perform unnecessary operations, such as tool switching, environment configuration and manual screening. Furthermore, referring to the detailed report, they can clearly find most of the security vulnerabilities hidden in the contract and fix them according to the modification suggestions.

In addition to the nine types of vulnerabilities listed in the table, our framework also supports the detection of other 45 common security issues, such as untrusted delegatecall, weak sources of randomness, etc., which basically covers almost all the vulnerability types supported by the state-of-the-art smart contract security testing tools. We can also integrate more advanced detection tools in the future.

5.5 Real Case Studies

5.5.1 Security-Reinforced Code Suggestion. When we type "Space", there will be a list of possible next-words shown on the screen. In order to fit the thinking mode of developers, our model needs to recommend a series of legal code pieces with various types or structures for selection. Take Fig. 3 as an example, it will display a list of possible tokens based on current context and sentence structure. When the length of a single candidate word is long and pre-defined variables have multiple common prefixes, this kind of suggestion can save a lot of time.

Table 9: Vulnerabilities detected by each tool. Our framework integrates those tools to facilitate the detection capability.

Category	Files	Vulns	Oyente	Securify	SmartCheck	Pied-Piper	Remix(Mythril)	Ours
Access Control	17	19	0 (0%)	0 (0%)	2 (11%)	0 (0%)	4 (21%)	4 (21%)
Arithmetic	15	24	17 (70.8%)	0 (0%)	3 (12.5%)	0 (0%)	14 (58.3%)	17 (70.8%)
Backdoor Threats	50	50	0 (0%)	0 (0%)	0 (0%)	48 (96%)	0 (0%)	48 (96%)
Front Running	4	7	0 (0%)	4 (57.1%)	0 (0%)	0 (0%)	4 (57.1%)	4 (57.1%)
Locked Ether	2	2	0 (0%)	0 (0%)	2 (100%)	0 (0%)	1 (50%)	2 (100%)
Reentrancy	31	32	29 (90.6%)	29 (90.6%)	29 (90.6%)	0 (0%)	29 (90.6%)	29 (90.6%)
Timestamp Dependency	5	7	0 (0%)	0 (0%)	2 (28.6%)	0 (0%)	0 (0%)	2 (28.6%)
Unchecked Low Calls	53	78	0 (0%)	69 (88.4%)	70 (89.7%)	0 (0%)	72 (92.3%)	72 (92.3%)
Unhandled Exception	4	7	7 (100%)	0 (0%)	0 (0%)	0 (0%)	4 (57.1%)	7 (100%)
Total	181	226	53 (23.5%)	102 (45.1%)	108 (47.8%)	48 (21.2%)	128 (56.6%)	185 (81.9%)

```
10    modifier isOwner(address _caller) {
11        require(msg.sender == factory);
12        require(_caller == owner);
13        _;
14    }
15
16    function increment(address caller)
17                                        isOwner
18    }                                   returns
                                          public
                                          auth
                                          {
```

Figure 3: Code example of predictions for next-words.

Different from other programming languages, smart contracts have more strict requirements on security. Therefore, ensuring the correctness and rigour of internal logic is an important part during code completion. Due to the limited space, we merely take arithmetic operation as an example to illustrate the advantages and necessity of security-reinforced code recommendation. Considering the situation in Fig. 4, when the developer declares two variables, one of his next possible operations is to get the sum. In the list of recommended tokens, the corresponding one is $x.add(y)$. Our framework calls the *add* function in the SafeMath library to implement the sum operation, avoiding the integer overflow vulnerability. Without reinforcement, the following prediction list may contain a simple "+" operation, which lacks the legitimacy check of result, and will lead to an overflow bug when operands are large.

```
21  contract Calculator {
22      uint256 result;
23
24      function getSum(uint256 x, uint256 y) public returns (uint256) {
25          uint256 a =
26      }                   x
27  }                       y
28                          x.add(y)                (method) using SafeMa…
                            x.sub(y)
                            result
```

Figure 4: Code example of predictions using safe functions.

5.5.2 Security-Oriented Code Validation. We choose two cases to demonstrate the security validation function. The first one is a smart contract with reentrancy vulnerability. The source code is shown in Fig. 5. It gives two security suggestions for this contract. At line 5 of the contract, the function makes an external call which is dangerous because the malicious contract can call back into this function and causes the different invocations of the function to interact in undesirable ways. This kind of vulnerability leads to *The DAO* event. As shown in line 7, there is also a low-level warning which indicates the keyword *throw* has been deprecated in the current version.

```
1   contract SendBalance {
2       mapping ( address => uint ) userBalances ;
3       bool withdrawn = false ;
4       function withdrawBalance (){
5           if (!( msg.sender.call.value      Diagnostic 1/2: Name: Reentrancy
6           (userBalances [msg.sender ])()))
7               throw;      Warning: Name: Deprecated constructions
8           userBalances [msg.sender ] = 0;
9       }}
10
```

Figure 5: An contract example with reentrancy threats.

The next case is taken from the source code of Soarcoin [38] contract, as shown in Fig. 6. Function *zero_fee_transaction* has been considered as a backdoor that can transfer out other account's balances arbitrarily. Although this kind of bug is not included in the SWC list, it has already been assigned with a CVE-ID: CVE-2018-1000203. Therefore, we set the severity level to "Error" to make developers pay attention to this high-privilege function.

```
11      function zero_fee_transaction(      Error: Name: ArbitraryTransfer problem
12          address _from,
13          address _to,
14          uint256 _amount
15      ) public onlycentralAccount returns(bool success) {
16          if (balances[_from] >= _amount &&
17              _amount > 0 &&
18              balances[_to] + _amount > balances[_to]) {
19              balances[_from] -= _amount;
20              balances[_to] += _amount;
21              return true;
22          } else {
23              return false;
24          }
25      }
26  }
27
```

Figure 6: Snippets taken from Soarcoin contract. Function *zero_fee_transaction* is an Arbitrary Transfer backdoor.

In summary, compared with Remix, the most widely used smart contract development platform, our framework achieves a higher accuracy in security-reinforced next-word recommendation, especially in the case of long context dependence, and does not need to input any prefix for recommendation. As for security analysis,

Remix depends on the MythX plug-in, whose core is Mythril, while our framework integrates five advanced contract analysis tools with unified execution and supports interactive debugging. It uses a variety of technologies to comprehensively analyze contracts and can expose common threats.

6 LESSONS LEARNED

During the design and implementation of the framework, we found out the following lessons worthy to be discussed:

The industry urgently needs an integrated development environment for smart contracts. In practice, we found that, unlike traditional software, which has specific development environments such as PyCharm, Visual C++ and Eclipse, there are only some simple editors for smart contracts, and they only support basic functions such as highlighting, word completion based on fuzzy search, etc. With the wide application of smart contract, a more intelligent and user-friendly development environment is needed to assist domain experts and engineers to implement the requirements more efficiently.

Security is an important indicator of smart contract, and the development environment needs targeted support. Due to the immutability of block-chain data, the smart contract cannot be modified after deployment. In other words, no patches can be applied to security issues in deployed contracts. In order to avoid property damage, developers need to keep paying attention to the security of the written words during the coding process, and have to use a variety of tools for bug finding. At present, code completion for smart contracts is based on fuzzy search, so that the security of code is not considered in the recommendation process. Although there are many security auditing tools for smart contracts, they support different types of vulnerabilities and require a lot of configurations before running. Therefore, a unified security-oriented coding and testing platform is urgently needed.

The degree of similarity between the training dataset and real-world usage scenarios will greatly affect the performance of code suggestion. Users of smart contracts are from all professions and trades, and engineers need to write various types of contracts, such as games, protocols or crowdfunding. In order to provide more reasonable code recommendation in different situations, the training dataset needs to be diverse enough and similar to the actual engineering code. Our model takes real-world contracts deployed on Ethereum as the samples, collects domain specific information and learns the general mode after reinforcement, which performs well in different scenarios.

7 RELATED WORK

Code Completion. Code completion technology is the most common program automation technology and is an important part of modern IDEs. Many works that explore the application of statistical learning and sequence models focus on the code completion task. Hindle et al. [18] first proposed probabilistic modeling of code token sequences in 2012. In 2014, on the basis of Hindle's research, Tu et al. [42] added a "cache" mechanism to maintain the program's locality based on the language model. There are also some works apply probabilistic grammars [3, 6] to the code completion task. Using structural information can improve the accuracy of code completion. Researchers usually conduct research in two directions: transforming the tree structure into sequences or directly modeling the tree structure. Li et al. [24] and Liu et al. [25] use the AST sequence to predict the terminal and non-terminal nodes of the program. This method of predicting the next tokens based on the AST sequence can not only predict the next terminal node, but also the structural information of the code, that is, the non-terminal node. Recently, Li et al. [20, 45] proposed Deep-AutoCoder to deal with the massive identifiers and another dual training framework to exploit the duality between code summarization and code generation tasks, improving the precision and efficiency greatly.

Smart Contract Vulnerability Detection. Smart contracts have been shown to be exposed to severe vulnerabilities [4, 19], and many efforts have been devoted to ensuring the correctness. There are many studies that provide security assurance for smart contracts in different ways. Some of them analyzed the EVM bytecode to check possible vulnerabilities during execution. For example, Luu et al. [27] designed Oyente, which builds the control-flow graph from the EVM bytecode and then performs symbolic execution and checks whether there exist potential vulnerable patterns. Static analysis tools often convert the contract code into a specific intermediate expression, and then perform pattern matching to find potential loopholes. For example, SmartCheck [40] uses XML-based intermediate representation to match vulnerabilities' patterns. Zeus [23] is another sound analyzer that translates smart contracts to the LLVM framework and uses XACML as a language to write properties. There are also some dynamic tools focus on smart contract vulnerability detection, like ContractFuzzer [21] and Echidna [16], which generate fuzzing inputs to trigger security vulnerabilities. Others carried out researches from the perspective of contract execution. Malicious behavior will be blocked from EVM layer through analyzing opcode sequences, like EVM* [28] and Sereum [35].

8 CONCLUSION

In this paper, we proposed an integrated framework which aims to make smart contract development more secure and easier. First, we implemented contract reinforcement based on AST and Datalog, and designed security-reinforcement code suggestion for Solidity language. Then, we integrated the security-oriented code validation with five free and open-source vulnerability detection tools, so that it can conduct a comprehensive analysis of the contract from different perspectives and cover Top-9 categories of high-risk security issues. During evaluation, our framework outperforms the most widely used platform and other professional vulnerability detection tools, which better guarantees the security of the entire process of contract development.

ACKNOWLEDGEMENT

We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This research is sponsored in part by the NSFC Program (No. 62022046), National Key Research and Development Project (Grant No. 2019YFB1706203), and the Webank-Tsinghua Smart Contract Security-Assured Platform Research Project (No. 20202000347).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. 2016. TensorFlow: A system for large-scale machine learning. *CoRR* abs/1605.08695 (2016). arXiv:1605.08695 http://arxiv.org/abs/1605.08695

[2] Maher Alharby and Aad Van Moorsel. 2017. Blockchain-based smart contracts: A systematic mapping study. *arXiv preprint arXiv:1710.06372* (2017).

[3] Miltiadis Allamanis and Charles A. Sutton. 2014. Mining idioms from source code. *ArXiv* abs/1404.0417 (2014).

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2016. A survey of attacks on Ethereum smart contracts. *IACR Cryptology ePrint Archive* 2016 (2016), 1007.

[5] Massimo Bartoletti and Livio Pompianu. 2017. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *International conference on financial cryptography and data security*. Springer, 494–509.

[6] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *ICML*.

[7] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *ArXiv* abs/1809.03981 (2018).

[8] François Chollet. 2019. Keras. https://keras.io/. Accessed November 4, 2019.

[9] ConsenSys. 2019. Mythril. https://github.com/ConsenSys/mythril-classic.

[10] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2015. Lab: Step by Step towards Programming a Safe Smart Contract. (2015).

[11] Docker. 2013. Empowering App Development for Developers. www.docker.com/.

[12] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2019. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. arXiv:1910.10601 [cs.SE]

[13] ethereum. 2019. Remix. http://remix.ethereum.org.

[14] Etherscan. 2019. Etherscan. https://etherscan.io/.

[15] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL* 2 (2018), 116:1–116:27.

[16] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 557–560.

[17] Erik Hillbom and Tobias Tillström. 2016. *Applications of smart-contracts and smart-property utilizing blockchains*. Master's thesis.

[18] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *ICSE 2012*.

[19] Yoichi Hirai. 2016. Formal verification of Deed contract in Ethereum name service. *November-2016.[Online]. Available: https://yoichihirai. com/deed. pdf* (2016).

[20] Xing Hu, Rui Men, Ge Li, and Zhi Jin. 2019. Deep-AutoCoder: Learning to Complete Code Precisely with Induced Code Tokens. In *43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15a-19, 2019, Volume 1*, Vladimir Getov, Jean-Luc Gaudiot, Nariyoshi Yamai, Stelvio Cimato, J. Morris Chang, Yuuichi Teranishi, Ji-Jiang Yang, Hong Va Leong, Hossain Shahriar, Michiharu Takemoto, Dave Towey, Hiroki Takakura, Atilla Elçi, Susumu Takeuchi, and Satish Puri (Eds.). IEEE, 159–168. https://doi.org/10.1109/COMPSAC.2019.00030

[21] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018* (2018). https://doi.org/10.1145/3238147.3238177

[22] juanfranblanco. 2019. vscode-solidity. https://github.com/juanfranblanco/vscode-solidity.

[23] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS*.

[24] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. *ArXiv* abs/1711.09573 (2018).

[25] Chang Liu, Xin Wang, Richard Shin, Joseph E. Gonzalez, and Dawn Song. 2017. Neural Code Completion.

[26] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).

[27] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. *IACR Cryptology ePrint Archive* 2016 (2016), 633.

[28] F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, H. Li, and X. Shi. 2019. EVM*: From Offline Detection to Online Reinforcement for Ethereum Virtual Machine. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 554–558.

[29] MythX. 2019. Smart Contract Weakness Classification and Test Cases. https://swcregistry.io/. Accessed November 4, 2019.

[30] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *CoRR* abs/1802.06038 (2018).

[31] OpenZeppelin. 2019. SafeMath. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol.

[32] Terence Parr. 2017. ANTLR. https://www.antlr.org/. Accessed November 4, 2017.

[33] Chao Peng, Sefa Akca, and Ajitha Rajan. 2019. SIF: A Framework for Solidity Contract Instrumentation and Analysis. *2019 26th Asia-Pacific Software Engineering Conference (APSEC)* (2019), 466–473.

[34] Daniel Perez and Benjamin Livshits. 2019. Smart Contract Vulnerabilities: Does Anyone Care? arXiv:1902.06710 [cs.CR]

[35] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. (2018).

[36] Sara Rouhani and Ralph Deters. 2019. Security, performance, and applications of smart contracts: A systematic survey. *IEEE Access* 7 (2019), 50759–50779.

[37] smartbugs. 2020. smart contracts dataset. https://github.com/smartbugs/smartbugs-wild.

[38] SoarLab. 2019. SoarCoin. https://etherscan.io/address/0xD65960FAcb8E4a2dFcb2C2212cb2e44a02e2a57E#code. Accessed November 4, 2019.

[39] Solidity. 2018. Solidity Programming Language. https://git.io/vFA47/.

[40] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, and Yaroslav Alexandrov. 2018. SmartCheck: static analysis of ethereum smart contracts. In *the 1st International Workshop*.

[41] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *ACM Conference on Computer and Communications Security*.

[42] Zhaopeng Tu, Zhendong Su, and Premkumar T. Devanbu. 2014. On the localness of software. In *FSE 2014*.

[43] Tsinghua University. 2019. Pied-Piper: Revealing the Backdoor Threats in Smart Contracts. https://github.com/renardbebe/BackdoorDetector.

[44] Various. 2018. GitHub - ethereum/solidity: The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity.

[45] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code Generation as a Dual Task of Code Summarization. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 6559–6569. http://papers.nips.cc/paper/8883-code-generation-as-a-dual-task-of-code-summarization

[46] yakindu. 2019. solidity-ide. https://yakindu.github.io/solidity-ide/.

[47] Shu Zhang, Dequan Zheng, Xinchen Hu, and Ming Yang. 2015. Bidirectional long short-term memory networks for relation classification. In *Proceedings of the 29th Pacific Asia conference on language, information and computation*. 73–78.