ISSN 2409-2665
Journal of Logistics, Informatics and Service Science
Vol. 9 (2022) No.1, pp.36-52
DOI:10.33168/LISS.2022.0104

A Smart Contract Weakness and Security Hole Analyzer Using Virtual Machine Based Dynamic Monitor

Yunsik Son [1] and YangSun Lee [2]

[1]Dongguk University, Seoul, Rep. of Korea
[2]SeoKyeong University, Seoul, Rep. of Korea

yslee@skuniv.ac.kr

Abstract. Blockchain-based smart contracts are a technology for decentralized applications, and their usefulness and development potential have been highly evaluated. However, as a technology developed over an extremely short period, there are many flaws in the programming language and execution environment used for developing and executing smart contracts. In this study, a software weakness analyzer is proposed to detect possible software weaknesses in smart contracts. The proposed analyzer examines software weaknesses through a security-level information flow. For this purpose, the semantics of a smart contract when converted into an intermediate code are defined and analyzed on a dynamic monitor.

Keywords: Blockchain, smart contract, software weakness, semantics, security-level information flow

1. Introduction

As the sizes of blockchain services and their markets continue to increase globally, various blockchain-based frameworks are entering the spotlight. Various blockchain-based frameworks have been developed, and it has become possible to use a general-purpose programming language when developing decentralized applications, thereby increasing the accessibility of decentralized application development. However, such accessibility causes security problems because security threats, such as logical errors, bugs, and mistakes, can inherently occur during development. Software vulnerabilities are classified as the main cause of cyberattacks in a cyber-physical infrastructure, which is a core component of modern society. Among the techniques used to defend against software vulnerabilities, the most effective method known is to remove the inherent weaknesses of the software in advance, such as through a secure development life cycle, and related research is also being carried out. A blockchain-based smart contract is a technology for decentralized applications, and its usefulness and development potential have been highly evaluated. The technology was developed over an extremely short period of time, and there are many flaws in its programming language and execution environment. As a result, significant damage, such as a DAO hacking incidents, have occurred (Abdellatif et al., 2018, and Atzei et al., 2017).

To effectively analyze smart contracts, the proposed technique translates the source code into the Smart Intermediate Language (SIL), which is an intermediate code for research, and analyzes vulnerabilities while executing it in a virtual machine (Lee et al., 2017, and Son et al., 2020). To clarify this process, in this study, for a security-level analysis of SIL, the security level of the variable of interest within the program is systematically identified by defining the formal semantic structure and evaluation rules.

2. Related studies

2.1. Dynamic program analysis

A dynamic program analysis is a computer software analysis performed through programs executed on real or virtual processors. For a dynamic program analysis to be effective, it must be run based on sufficient test input data to contain virtually any output that the analyzed program can create. This dynamic program analysis is used for code coverage or memory error detection according to the workload, fault localization for finding buggy codes according to the failure or passing of test cases, and a security-level analysis for detecting security problems (Kim et al., 2014, and Lee et al., 2017). In this study, a security weakness analysis is conducted based on a dynamic analysis of the semantic structure of the intermediate language defined for a security-level flow analysis during the dynamic monitoring of a virtual machine.

2.2. Security-level information flow analysis

A security-level information analysis is a technique that analyzes the propagation of information disclosed by a system and is an analysis technique that dynamically tracks the information flow at runtime. In general, in a security-level information flow analysis, each variable is assigned a security level, the basic model of which consists of two distinct levels, low and high, or in other words, publicly available and confidential information. To ensure confidentiality, the security applied should not allow information to propagate from high-security variables to low-security variables, and to ensure the integrity, should restrict information propagation to high-security variables.

In this study, we propose a smart contract weakness analysis technique that applies a security-level information flow. In general, to obtain such information, it is necessary to partially process the same task as the execution process of the program because the larger the analysis target range, the greater the complexity and the need for a number of path analyses. Therefore, in most flow analyses, the scope is limited to the main area of interest, or even to the basic block unit (Bhargavan et al., 2016). Unlike with existing methods, this study defines the formal semantics for the intermediate code and directly monitors possible weaknesses using a virtual machine.

2.3. Hyperledger fabric

Hyperledger fabric is a licensed blockchain network provided by IBM and Digital Asset, and is a platform for developing blockchain solutions and applications (Cachin, 2016). It provides a modular architecture representing the role between nodes in a blockchain network, the execution of smart contracts (chain code of the fabric), and a configurable consensus and membership services.

Hyperledger fabric blockchain networks execute the chain code, access the ledger data, approve transactions, and interface with applications. This is in contrast to traditional blockchain platforms, where smart contracts must be written in domain-specific languages or rely on cryptocurrency (Jeong et al., 2021, and Zheng et al., 2018). Figure 1 shows the structure of the smart contract on hyperledger fabric.

Because the chain code running on a hyperledger fabric network cannot be arbitrarily modified when the contract is completed, it can develop into a security vulnerability when a chain code with a security weakness is executed. To solve this problem, it is therefore necessary to diagnose security weaknesses using static analysis methods that can be analyzed prior to software execution (Abdellatif et al., 2018, Bhargavan et al., 2016, and Lin et al., 2017).

2.4. Smart intermediate language for smart virtual machine of runtime environments

Smart Intermediate Language (SIL), the virtual machine code for Smart Virtual Machine (SVM) of runtime environments, is designed as a standardized virtual machine code model for ordinary smart devices and embedded systems. SIL is a

stack-based command set that is independent of language, hardware, and platform (Lee et al., 2017, and Son et al., 2020). To accommodate a variety of programming languages, SIL is defined based on an analysis of existing virtual machine codes, such as bytecode and .NET IL, among others. In addition, it also has a set of arithmetic operation codes to cover procedural programming languages and object-oriented languages.

SIL is composed of a meta-code that carries out particular jobs, such as class creation, and an operation code that responds to actual commands. An operation code has an abstract form that is not subordinate to specific hardware or source languages. It is defined in mnemonic to heighten the readability and applies a consistent name rule to make debugging in assembly language levels easier to achieve. In addition, it has a short-form operation code for optimization. SIL has six groups (excluding the optimization group) of operation codes, and Fig. 2 shows the category of SIL operation codes.

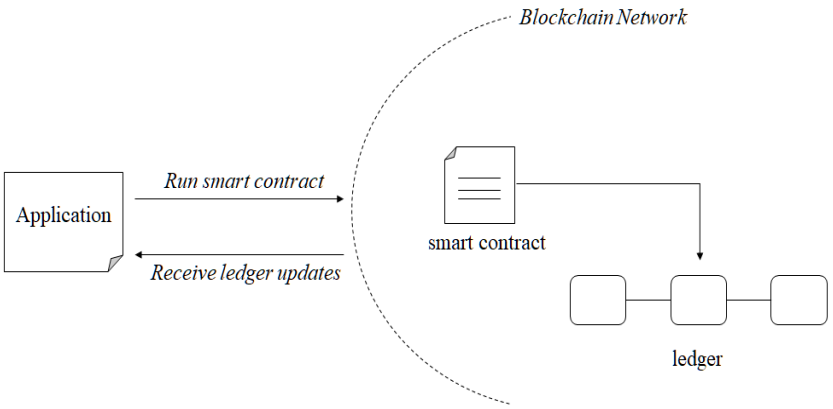


Fig. 1: Structure of a smart contract on hyperledger fabric

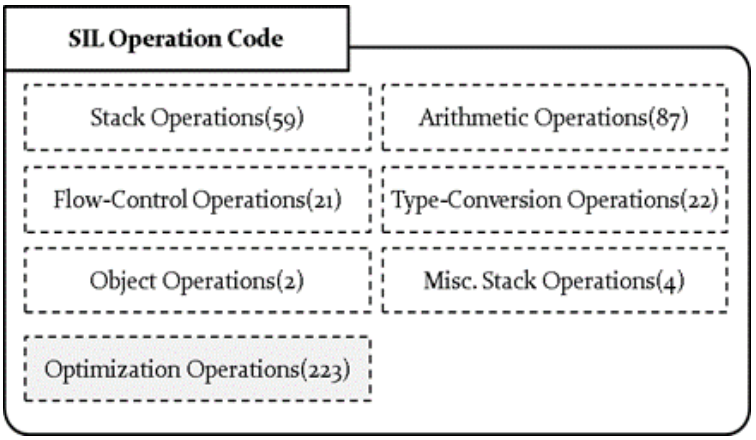


Fig. 2: Category of SIL operation codes

2.5. Smart virtual machine (SVM) of runtime environments

SVM's detailed module configuration is illustrated in Fig. 4. Largely, it combines five components: an Smart Executable file Format (SEF) loader, which is a load input SEF file in memory; an interpreter for a stack-based evaluation of the instructions in memory ; a managing module group for a runtime environment; a built-in SVM library; and a native interface, which is used for interactions with the native platform. It is also designed for additional components, such as a debugging and profiling interface.

The interpreter is the core SVM module, and is the SIL code execution routine from the loaded SEF file. The interpreter has action procedures that are mapped to each SIL code, and it executes instructions with a reference, i.e., metadata stored by the loader.

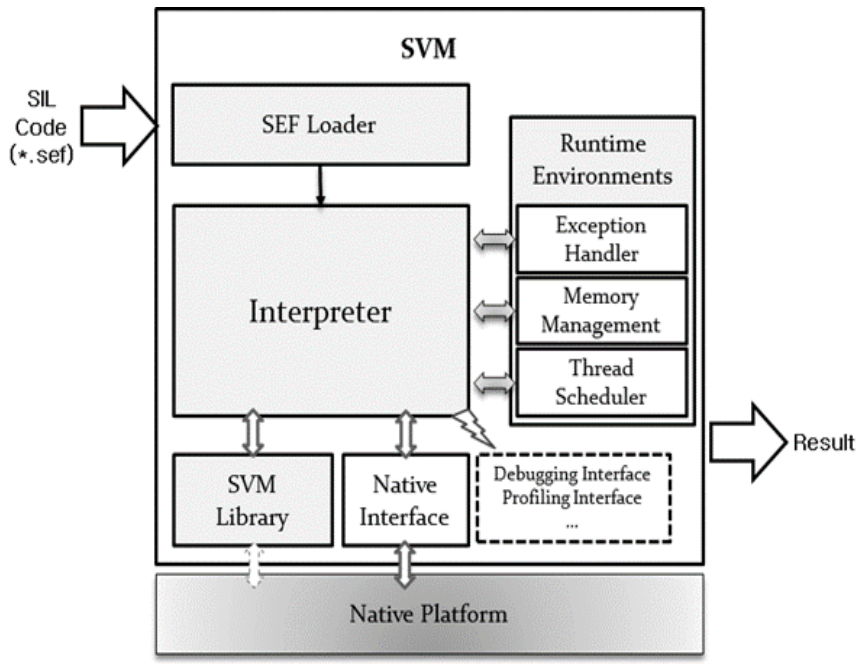


Fig. 3: System configuration of smart virtual machine

During execution, the evaluated data are stored and managed in a stack or heap, and if an error occurs while executing, the exception handler catches the error and outputs the related error message, halting the VM instance for the given program (Lee et al., 2017, and Son et al., 2020).

2.6. Weakness analysis method using dynamic monitor

2.6.1. Weakness analysis for security-level information

In this study, we examine a method for detecting weaknesses that occur when an external input value is used without a proper validation using a data flow analysis of the variable. For this, each variable is assigned security-level information and its use is monitored. In the proposed weakness analysis method, security-level information is defined in two ways.

Low: When the value of the variable is defined by an unreliable external variable or is influenced by a low-level variable.

High: When the value of the variable is not defined externally or is validated by an arbitrary function.

The process for analyzing the security level of a variable in a smart contract based on the above criteria is as follows: First, the program is divided into a declaration part and a sentence part . The initialization part of the variable information is analyzed in the declaration part, and the use and change process of the variable information are analyzed in the sentence. The initialization of the variable information in the declaration part is classified as follows:

1) Parameters of functions and other variables, or initialization by function return value
2) Initialization by constant value
3) Initialization by arithmetic expression

In general, the initialization process has the form x = y, and in this case, the security level of y is propagated to x. Therefore, initialization by the return value of other variables, including parameters or functions, depends on the security level of the corresponding value. In addition, initialization using a constant value sets a high value, and initialization using an arithmetic expression depends on the evaluation of the corresponding arithmetic expression. Statements in which variables are used are generally classified into three types: arithmetic expressions, conditional branching statements, and looping statements. The security-level information of the operation result is determined by using the security level of the operands as the basic unit for analyzing the security level in the operation expression. The operation process is analyzed as a postfix, and the security level determined in each operation process is used as the security level of the upper operation expression.

The security-level information for the basic operational process is shown in Fig. 4. When an expression x + y exists, if an operand has low security-level information, the result of the expression is low security-level information until the validation function is applied.

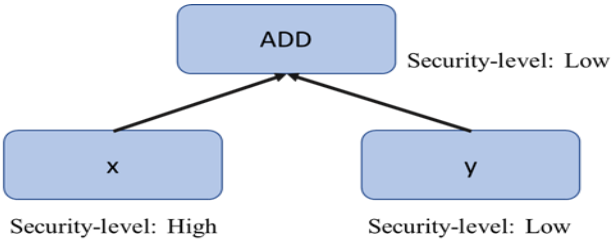


Fig. 4: Security-level information propagation

In condition/branch statements, the sentences comprising each expression are composed of basic blocks, and each basic block is connected to a control flow. Using the connected control flow, the process of changing the value of the variable used in each sentence is traced within the flow. Figure 5 shows a flow graph for a statement in which "if ~ else" statements are nested. The part grouped by the dotted line is the unit in the branching process, and the part indicated by the solid line is the part where the actual sentence can be executed. An analysis should be conducted for each execution flow.

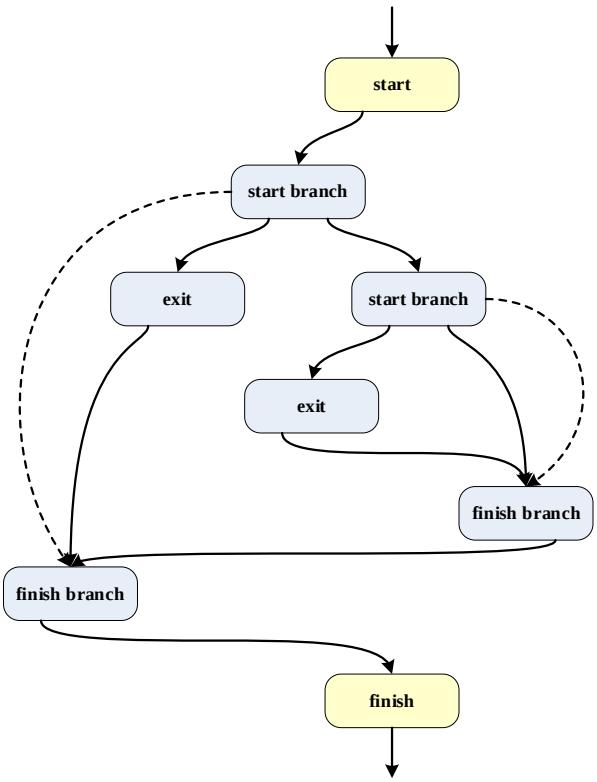


Fig. 5: Execution path example of nested branch statement

Table 1: Definitions of sematic object

$$\boldsymbol{b = base}$$

$$\boldsymbol{o = offset}$$

$$\boldsymbol{a = address}$$

$$\boldsymbol{v \in V \quad Value}$$

$$\boldsymbol{l \in L \quad Security\ level\ of\ v}$$

$$\boldsymbol{x \in X\ (v, l)}$$

$$\boldsymbol{V = set\ of\ value}$$

$$\boldsymbol{L = Set\ of\ security\ level} ::= \{\boldsymbol{low, high}\}$$

$$\boldsymbol{X = Pair\ set\ of\ valuse\ and\ security\ level} ::= \boldsymbol{V \times L}$$

$$\boldsymbol{A = Secure\ activation\ record} ::= \{\boldsymbol{x_0, x_1, x_2, \dots, x_x}\}$$

$$\boldsymbol{A_i = Set\ of\ elements\ of\ A\ with\ base\ i} ::= \{\boldsymbol{x_0, x_1, x_2, \dots, x_x}\}$$

$$\boldsymbol{A_i} = \begin{cases} \boldsymbol{global\ record} & \boldsymbol{if\ i = 0} \\ \boldsymbol{function\ record} & \boldsymbol{if\ i \geq 1} \end{cases}$$

$$\boldsymbol{S = Secure\ operation\ stack} ::= \{\boldsymbol{x_0, x_1, x_2, \dots, x_x}\}$$

2.7. Semantics definition of smart intermediate language for dynamic monitor

If the security level of a variable is low, the program may be at risk if the information is used for sensitive tasks, such as security decisions or manipulation of the DB and transaction values. The following defines the semantic structure used to analyze the previously defined security-level information during the program execution.

The execution of the smart contract is applied in SVM, and the semantic structure for security-level analysis is defined for SIL, which is an intermediate language of the SVM. Based on the defined semantic structure, the security level of the variable of interest is monitored by dynamically analyzing the security-level information flow.

To define the semantic structure of the specific SIL for a security-level information analysis, the semantic object is defined, as shown in Table 1. A semantic object is a set of objects required for evaluation rules and SIL security level analysis, and consists of data structures, values, and security levels used for an information flow analysis.

The proposed analysis method monitors changes in the environmental information according to the execution of the commands. The environmental information is defined as a secure activation record A and an operation stack S. Because the security activation record maintains the information of the variable and

security-level information of the corresponding variable, it is possible to monitor the security-level information of the variable according to the execution of the command. The command evaluation rules are expressed as follows.

$$A, S \vdash I \Downarrow A', S'$$

In this study, the evaluation rule is defined in the form of an inference rule using environmental information and stack manipulation functions. Table 2 lists the environmental information and stack manipulation functions applied in the evaluation rules.

The evaluation rules are largely divided into stack instruction evaluation rules and arithmetic instruction evaluation rules for each instruction. The security level of information is evaluated according to the instruction execution and returns an activation record with an operation stack that manages the execution information and security level. The evaluation rules in this study assume that the security level of constant values is high, and that the security level of the corresponding information increases when information is converted through the instructions. Tables 3 and 4 list the evaluation rules for a stack operation and an instruction, respectively.

Table 2: Manipulation functions for environments and stack

$\boldsymbol{Update} = \boldsymbol{A} \times (\boldsymbol{b}, \boldsymbol{o}) \times \boldsymbol{X} \rightarrow \boldsymbol{A'}$
$\boldsymbol{Get} = \boldsymbol{A} \times (\boldsymbol{b}, \boldsymbol{o}) \rightarrow \boldsymbol{X}$
$\boldsymbol{LoadIndirect} = \boldsymbol{a} \rightarrow \boldsymbol{X}$
$\boldsymbol{StoreIndirect} = \boldsymbol{a} \times \boldsymbol{X} \rightarrow \boldsymbol{A'}$
$\boldsymbol{Top} = \boldsymbol{S} \rightarrow \{\boldsymbol{x_{top}}\}$
$\boldsymbol{Top2} = \boldsymbol{S} \rightarrow \{\boldsymbol{x_{top}}, \boldsymbol{x_{top-1}}\}$
$\boldsymbol{Push} = \boldsymbol{S} \times \boldsymbol{X} \rightarrow \boldsymbol{S'}$
$\boldsymbol{Pop} = \boldsymbol{S} \rightarrow \boldsymbol{X}$

Table 3: Evaluation rules for stack operations

$$\textbf{dup}: \frac{A,\ S \vdash I \Downarrow A', S'}{A', S' \vdash dup \Downarrow A', S'.Push(S'.Top())}$$

$$\textbf{dup2}: \frac{A,\ S \vdash I \Downarrow A', S'}{A',\ S' \vdash dup \Downarrow A',\ S'.Push(S'.Top2())}$$

$$\textbf{ldc}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash ldc\ v \Downarrow A',\ S'.Push((v,high))}$$

$$\textbf{lod}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash lod\ b\ o \Downarrow A',\ S'.Push(A'.Get(b,o))}$$

$$\textbf{str}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash str\ b\ o \Downarrow A'.Update((b,o),S'.Pop()),S'}$$

$$\textbf{lda}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash lda\ b\ o \Downarrow A',\ S'.Push(\&A'.Get(b,o),\ high)}$$

$$\textbf{ldi}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash ldi \Downarrow A',\ S'.Push(LodIndirect(S'.Pop().v))}$$

$$\textbf{sti}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash sti \Downarrow StoreIndirect(S'.Pop().v,\ S'.Pop()),S'}$$

Table 4: Evaluation rules for instructions

$$\textbf{add}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash add \Downarrow A',\ S'.Push((S'.Pop().v+S'.Pop().v,\ high)}$$

$$\textbf{sub}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash sub \Downarrow A',\ S'.Push((S'.Pop().v-S'.Pop().v,\ high)}$$

$$\textbf{mul}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash mul \Downarrow A',\ S'.Push((S'.Pop().v*S'.Pop().v,\ high)}$$

$$\textbf{div}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash div \Downarrow A',\ S'.Push((S'.Pop().v/S'.Pop().v,\ high)}$$

$$\textbf{mod}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash mod \Downarrow A',\ S'.Push((S'.Pop().v\%S'.Pop().v,\ high)}$$

$$\textbf{neg}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash neg \Downarrow A',\ S'.Push((-S'.Pop().v,\ high)}$$

$$\boldsymbol{eq}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash eq \Downarrow A',\ S'.Push((S'.Pop().v == S'.Pop().v,\ high)}$$

$$\boldsymbol{ne}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash ne \Downarrow A',\ S'.Push((S'.Pop().v != S'.Pop().v,\ high)}$$

$$\boldsymbol{ge}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash ge \Downarrow A',\ S'.Push((S'.Pop().v \geq S'.Pop().v,\ high)}$$

$$\boldsymbol{gt}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash gt \Downarrow A',\ S'.Push((S'.Pop().v > S'.Pop().v,\ high)}$$

$$\boldsymbol{le}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash le \Downarrow A',\ S'.Push((S'.Pop().v \leq S'.Pop().v,\ high)}$$

$$\boldsymbol{lt}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash lt \Downarrow A',\ S'.Push((S'.Pop().v < S'.Pop().v,\ high)}$$

$$\boldsymbol{band}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash band \Downarrow A',\ S'.Push((S'.Pop().v \& S'.Pop().v,\ high)}$$

$$\boldsymbol{bor}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash bor \Downarrow A',\ S'.Push((S'.Pop().v \mid S'.Pop().v,\ high)}$$

$$\boldsymbol{bxor}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash bxor \Downarrow A',\ S'.Push((S'.Pop().v \wedge S'.Pop().v,\ high)}$$

$$\boldsymbol{bcom}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash bcom \Downarrow A',\ S'.Push((\sim S'.Pop().v,\ high)}$$

$$\boldsymbol{shl}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash shl \Downarrow A',\ S'.Push((S'.Pop().\ll S'.Pop().v,\ high)}$$

$$\boldsymbol{shr}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash shr \Downarrow A',\ S'.Push((S'.Pop().v \gg S'.Pop().v,\ high)}$$

$$\boldsymbol{and}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash and \Downarrow A',\ S'.Push((S'.Pop().v \&\& S'.Pop().v,\ high)}$$

$$\boldsymbol{or}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash or \Downarrow A',\ S'.Push((S'.Pop().v || S'.Pop().v,\ high)}$$

$$\textbf{not}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash \textbf{not} \Downarrow A',\ S'.\textbf{Push}((!S'.\textbf{Pop}().v,\ \textbf{high})}$$

$$\textbf{inc}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash \textbf{inc} \Downarrow A',\ S'.\textbf{Push}((++S'.\textbf{Pop}().v,\ \textbf{high})}$$

$$\textbf{dec}: \frac{A,\ S \vdash I \Downarrow A',\ S'}{A',\ S' \vdash \textbf{dec} \Downarrow A',\ S'.\textbf{Push}((--S'.\textbf{Pop}().v,\ \textbf{high})}$$

3. Experiment results

The overall structure of the proposed system is shown in Fig. 6.

When a smart contract is input, it is converted into abstract syntax tree at the front end, which is again converted into an intermediate language through an intermediate code converter. The intermediate language uses SIL, through which a security weakness analysis is applied.

A security weakness analysis is conducted using a separate analysis engine or an SVM with a dynamic monitor.

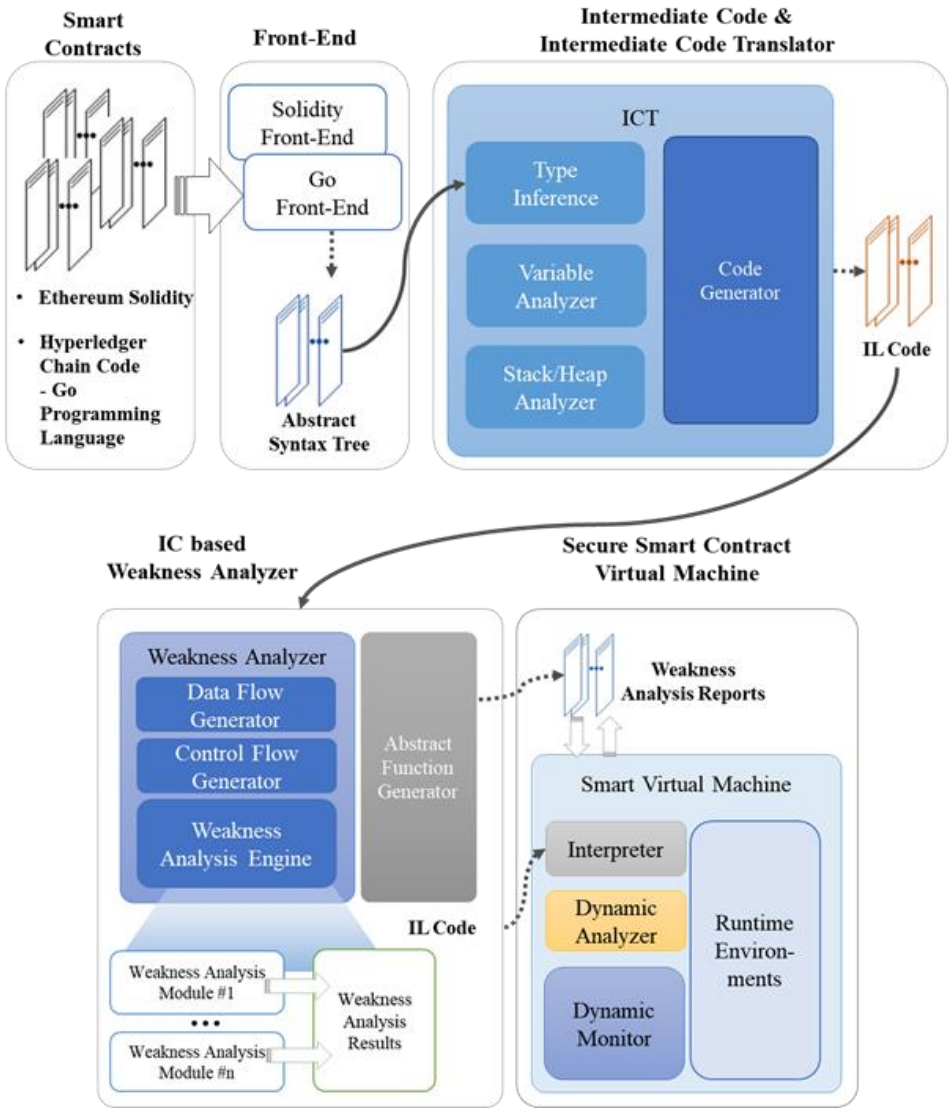
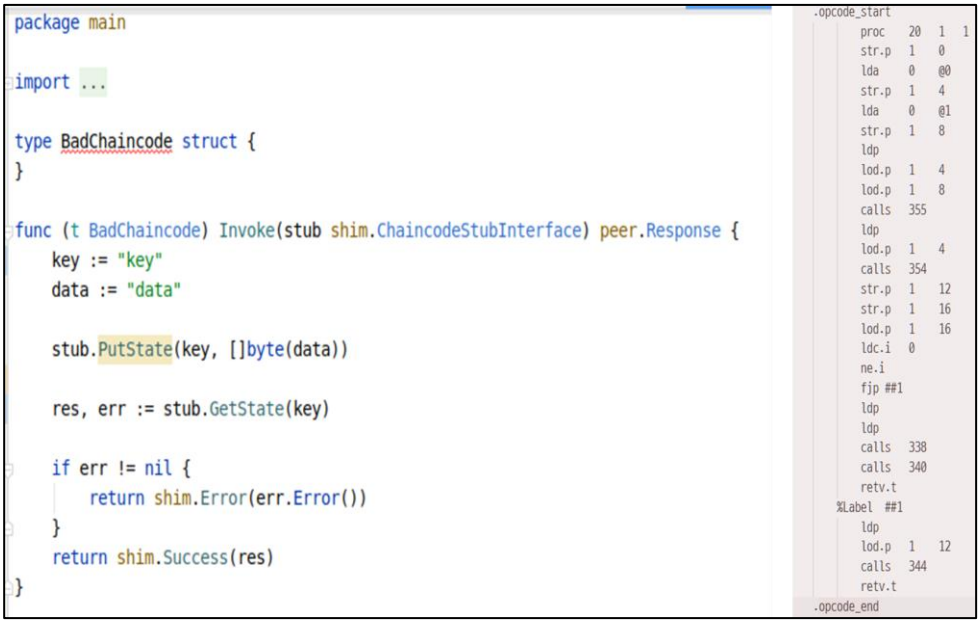


Fig. 6: Structure of the chaincode security weakness analyzer

In this study, a method for dynamically conducting a security weakness analysis using a dynamic monitor is proposed. For this, the semantic structure and evaluation rule of each command is defined, and in this way, the security level of the variable of interest is analyzed.

Figure 7 shows an example code that uses the value defined inside the function, stores it in the external state value, reads the value from the external state value again, and assigns it to the internal object. To guarantee the state value in the contract, a separate state management is required. Because there is no code for managing the state in this example, the value returned by the GetSate function must be verified.

When using Goroutines and Channels for concurrency programming, caution should be exercised because they can lead to non-deterministic behavior in smart contracts owing to issues regarding the input validation and race condition if not handled properly. The result of translating the example code into the SIL code is shown on the right side of Fig. 7.



```
package main

import ...

type BadChaincode struct {
}

func (t BadChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    key := "key"
    data := "data"

    stub.PutState(key, []byte(data))

    res, err := stub.GetState(key)

    if err != nil {
        return shim.Error(err.Error())
    }
    return shim.Success(res)
}
```

```
.opcode_start
        proc    20  1  1
        str.p   1   0
        lda     0   @0
        str.p   1   4
        lda     0   @1
        str.p   1   8
        ldp
        lod.p   1   4
        lod.p   1   8
        calls   355
        ldp
        lod.p   1   4
        calls   354
        str.p   1   12
        str.p   1   16
        lod.p   1   16
        ldc.i   0
        ne.i
        fjp ##1
        ldp
        ldp
        calls   338
        calls   340
        retv.t
    %Label  ##1
        ldp
        lod.p   1   12
        calls   344
        retv.t
.opcode_end
```

Fig 7: Smart contract example (Chaincode for Hyperledger Fabric) and translated SIL code

Figure 8 shows the process of analyzing the code in Fig. 7 through the semantic structure and evaluation rules for the security level information of the SIL defined in this paper. It can be seen that the level of security information varies according to the execution of the command for each variable, and it can be confirmed that the security level for the return value of the GetSate function is evaluated as low.

4. Conclusions and further researches

Smart contracts are decentralized applications that operate based on blockchain technology. If the programming language and execution environment used for the creation of smart contracts are not strictly defined, security weaknesses, which are the root cause of software vulnerabilities, may be inherent in the code and execution environment. Owing to the nature of blockchain, once a contract is completed and a block is created, the contract cannot be arbitrarily modified. Thus, if a chain code with a security weakness is contracted, it cannot be modified, causing a security threat. A smart contract must be analyzed for weaknesses before the contract is executed.

In this study, we proposed a method for analyzing the security level of variables and detecting security weaknesses in smart contracts. Unlike the existing static analysis method, the proposed method utilizes a dynamic monitoring of the virtual system to obtain the same analysis result as when executing the actual code, by which an evaluation rule that can analyze the security level in the semantic structure of the SIL code is added and formally defined. Thus, it is possible to systematically analyze security weaknesses. In the future, we plan to improve the algorithm for calculating the evaluation rules, allowing the formal semantic rules to be evaluated more effectively, and we expect that it will be possible to quickly conduct a dynamic analysis.

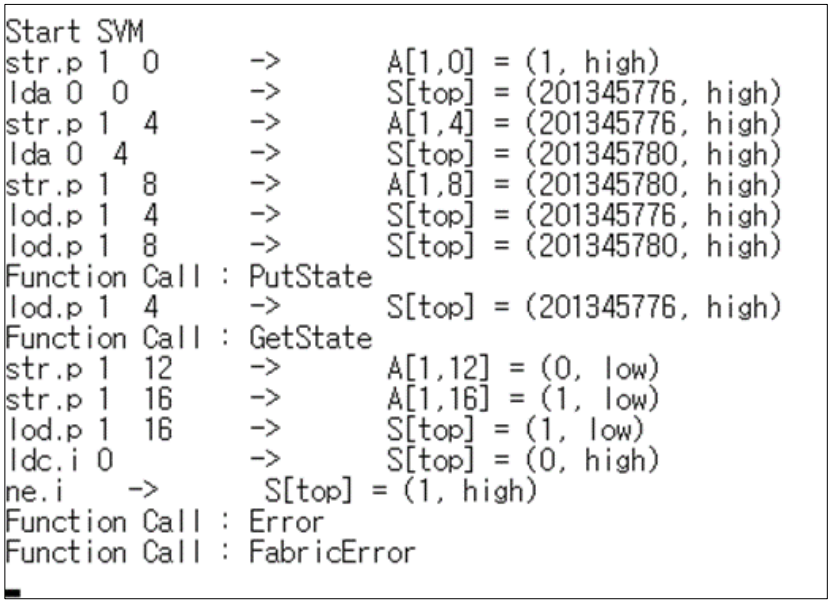


Fig. 8: Analysis results of smart contract example

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) (No.2019R1F1A1045343, No. 2020R1A2C1013296).

References

Abdellatif, T., & Brousmiche, K. L. (2018). Formal Verification of Smart Contracts based on Users and Blockchain Behaviors Models. *In 9th IFIP International Conference on New Technologies, Mobility and Security*, 1-5.

Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A Survey of Attacks on Ethereum Smart Contracts SoK. *In Proceedings of the 6th International Conference on Principles of Security and Trust*, (pp. 164-186).

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., & Zanella-Béguelin, S. (2016). Formal Verification of Smart Contracts. *In Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, (pp. 91-96).

Cachin, C. (2016). Architecture of the Hyperledger Blockchain Fabric, *IBM Research Report*, 1-4

Jeong, J., Kim, D., Ihm, S., Lee, Y., & Son, Y. (2021). Multilateral Personal Portfolio Authentication System Based on Hyperledger Fabric. *ACM Transactions on Internet Technology, 21(1),* 1-17.

Jeong, J., Lim, J., & Son, Y. (2019). A Data Type Inference Method Based on Long Short-Term Memory by Improved Feature for Weakness Analysis in Binary Code. *Future Generation Computer Systems, 100*, 1044-1052.

Kim, D., Ihm, S., & Son., Y. (2021). Two-Level Blockchain System for Digital Crime Evidence Management, *Sensors, 21(9),* 3051, 1-17.

Kim, J., & Lee, Y. (2014). A Study on the Optimization Method for the Rule Checker in the Secure Coding, *International Journal of Security and Its Applications, 8(1),* 333-342. http://dx.doi.org/10.14257/ijsia.2014.8.1.31

Lee, Y., Jeong, J., & Son, Y. (2017). Design and implementation of the secure compiler and virtual machine for developing secure IoT services. *Future Generation Computer Systems, 76*, 350-357.

Lin, I.C., & Liao, T.C. (2017). A Survey of Blockchain Security Issues and Challenges, *International Journal of Network Security, 19(5),* 653-659. http://dx.doi.org/10.6633/IJNS.201709.19(5).01

Nasir, Q., Qasse, I.A., Abu Talib, M., & Nassif, A.B. (2018). Performance Analysis of Hyperledger Fabric Platforms. *Security and Communication Networks, 2018*, 1-14, https://doi.org/10.3390/computers10010007

Park, J., Kim, H., Kim, G., & Ryou, J. (2020). Smart Contract Data Feed Framework for Privacy-Preserving Oracle System on Blockchain. *Computers, 10(1),* 1-12, doi:10.3390/computers10010007.

Son, Y., Jeong, J., & Lee, Y. (2020). Design and Implementation of an IoT-Cloud Converged Virtual Machine System, *The Journal of Supercomputing, 76(5)*, 5259-5275. https://doi.org/10.1007/s11227-019-02866-x

Zheng, Z., Xie, S., Dai, H.N., Chen, X., & Wang, H. (2018). Blockchain challenges and opportunities: A survey. *Internatioal Journal of Web and Grid Services, 14(4)*, 352-375.