A Study on Intermediate Code Generation for Security Weakness Analysis of Smart Contract Chaincode

Yangsun Lee
SeoKyeong University, Seoul, Rep. of Korea. E-mail: yslee@skuniv.ac.kr

Received September 28, 2021; Accepted December 20, 2021
ISSN: 1735-188X
DOI: 10.14704/WEB/V19I1/WEB19318

Abstract

The hyperledger fabric is a modular blockchain framework used by private companies to develop blockchain-based products, solutions, and applications using plug-and-play components. The smart contracts operating in this framework is created by implementing a chaincode. When implementing a chaincode, there may be a security weakness inside the code, which is the root cause of the security vulnerability. However, when the contract is completed and the block is created, the chaincode cannot be arbitrarily modified, so the security weakness must be analyzed before execution.

This paper conducted a study on chaincode intermediate code generation for security weakness analysis of chaincode operating in hyperledger fabric blockchain framework. Analysis of security weaknesses at the source code level is not easy because the code logic is not clear and the complexity is high. On the other hand, security weakness analysis at the intermediate code level is easy to analyze because the code logic of the source code is clearly represented and the complexity is lower than that of the source code.

Keywords

Security Weakness Analysis, Smart Contract, Chaincode, Hyperledger Fabric, Blockchain, Intermediate Code Generation

Introduction

The hyperledger fabric is a modular blockchain framework used by private companies to develop blockchain-based products, solutions, and applications using plug-and-play components. The smart contracts operating in this framework is created by implementing a chaincode. When implementing the chaincode, there may be a security weakness inside the code, which is the root cause of the security vulnerability, but when the contract is completed and a block is created, the chaincode cannot be modified arbitrarily, so the security weakness must be analyzed before execution (Lin, Liao, 2017. Fagan, 1976, Son, et al., 2015).

This paper conducted a study on chaincode intermediate code generation for security weakness analysis of chaincode operating in hyperledger fabric blockchain framework. Security weakness analysis includes static analysis and dynamic analysis techniques. Static analysis is a technique that analyzes a program without running it, and dynamic analysis is a technique that analyzes the program as it actually runs. Analysis of security weaknesses at the source code level is not easy because the code logic is not clear and the complexity is high. On the other hand, intermediate code clearly expresses the code logic of the source code, and because its complexity is lower than that of the source code, it is easy to analyze all the execution paths of the program. Dynamic analysis is also easy because intermediate code can be executed on virtual machine execution systems targeting intermediate code (Lee et al., 2017, Son, Lee, 2012, Jeong et al., 2019, Cousot, Cousot, 2002).

Related Studies

Hyperledger Fabric Framework

The hyperledger fabric (Cachin, 2016) is a licensed blockchain network provided by IBM and Digital Asset, a platform for developing blockchain solutions and applications. It provides a modular architecture that represents the role between nodes in a blockchain network, the execution of smart contracts(fabric's chaincode), and the configurable consensus and membership services. The hyperledger fabric blockchain networks execute chaincode, access ledger data, approve transactions, and interface with applications.

Since chaincode running on the hyperledger fabric network cannot be arbitrarily modified when the contract is completed, it can develop into a security vulnerability when the chaincode with security weakness is executed. Therefore, in order to solve this problem, it is necessary to diagnose security weakness items using static analysis methods that can be analyzed before software execution.

Software Security Weakness Analysis

A Software security weakness analysis is an analysis technique that diagnoses whether the security weakness, which is the root cause of security vulnerability, exists inside the software, and proactively detects and removes potential vulnerabilities such as software defects and errors to proactively eliminate the possibility of security threats such as hacking. Security weakness analysis method is divided into static analysis and dynamic analysis

(Kim et al., 2020, Wichmann et al., 1995, Cousot, Cousot, 2002, Abdellatif, Brousmiche, 2018, Bhargavan et al., 2016).

Static analysis is usually done by code review and is performed during the implementation phase of the security development life cycle. The ideal static analysis is to find software defects automatically. However, this increases time and resource costs. This helps security analysts find security weaknesses in their areas of interest, rather than automatically finding them. Unlike static analysis, dynamic analysis does not have access to the source code, and vulnerability scanning and penetration testing are used as dynamic analysis methods to find security weaknesses in running applications.

Intermediate Code

Intermediate code is an intermediate step code that is meaningfully equivalent to the source code independent of the target machine to help analyze the computer program. The intermediate code acts as an intermediate type of code that connects the front and rear ends of the compiler. Compilers have become functionally independent modules by using intermediate code, and their portability has increased (Lee et al., 2017, Son, Lee, 2012, Abdellatif, Brousmiche, 2018).

In addition, the translation process can be more easily expressed and efficiently processed because it serves as an intermediate between advanced source code and lower-level objective code, enabling machine-independent and more efficient optimization by using intermediate code. Using the intermediate language, the programs can be executed independently using a virtual machine of the target machine

Chaincode Intermediate Code Generator

Analysis of security weaknesses at the source code level is not appropriate due to the lack of clarity and high complexity of the code logic. On the other hand, the intermediate code is semantically equivalent to the source code, the code logic is clear and concise, and the complexity is low. In addition, the presence of intermediate code, a common format, can increase scalability by reusing security weakness modules, even if the target language is different.

This paper studies intermediate code generators that generate intermediate code meaningfully equivalent to chaincode for security weakness analysis inherent in chaincode. The intermediate code generator in this paper converts a chaincode into a pre-defined Smart

Intermediate Language (SIL) code to generate the Smart Assembly Format (SAF), which is an assembly file format (Lee et al., 2017, Son, Lee, 2012, Lee et al., 2020, Son, Lee, 2014). Figure 1 shows the structure of an intermediate code generator.

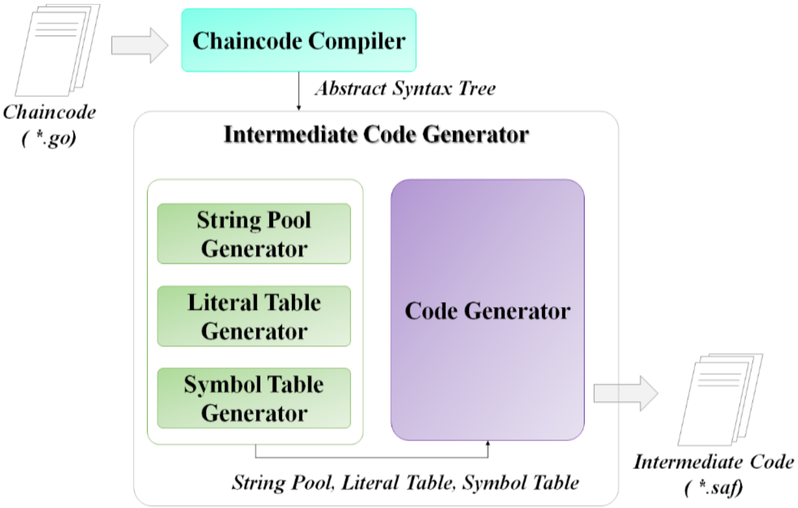


Figure 1. Structure of an Intermediate Code Generator

Information Table for Generating Intermediate Code

Intermediate code generation requires a code generation information. This paper generates a string pool, literal table, and symbolic table, which are information tables for code generation.

String Pool Generator

The string pool generator generates a symbol identification table for access to the symbol table when generating the intermediate code. The string pool consists of a symbol name and a symbol identifier, and it accesses the symbol table when generating the intermediate code.

String pool generation is created through an abstract syntax tree (AST) circulation. The goal is to create a table that identifies the name of the symbol, so when visiting the AssignStmt, ValueSpec, TypeSpec, and FuncDec nodes, where the symbol can exist among the AST nodes, symbol identifiers are generated.

Literal Table Generator

Literal table generators generate literal tables by collecting information to process string literals, escape sequences, and format specificators during intermediate code generation.

Literal table generation is generated while touring AST. Because the purpose of the string literal is to generate information, the generator generates literal information when it visits BasicLit among AST nodes.

When visiting the BasicLit node, the generator analyzes whether the literal is a string literal. If it is a string literal, it analyzes the existence of the escape sequences and type specifiers in the string literal, and inserts the string literal into the literal table. Figure 2 shows the literal table generation process.

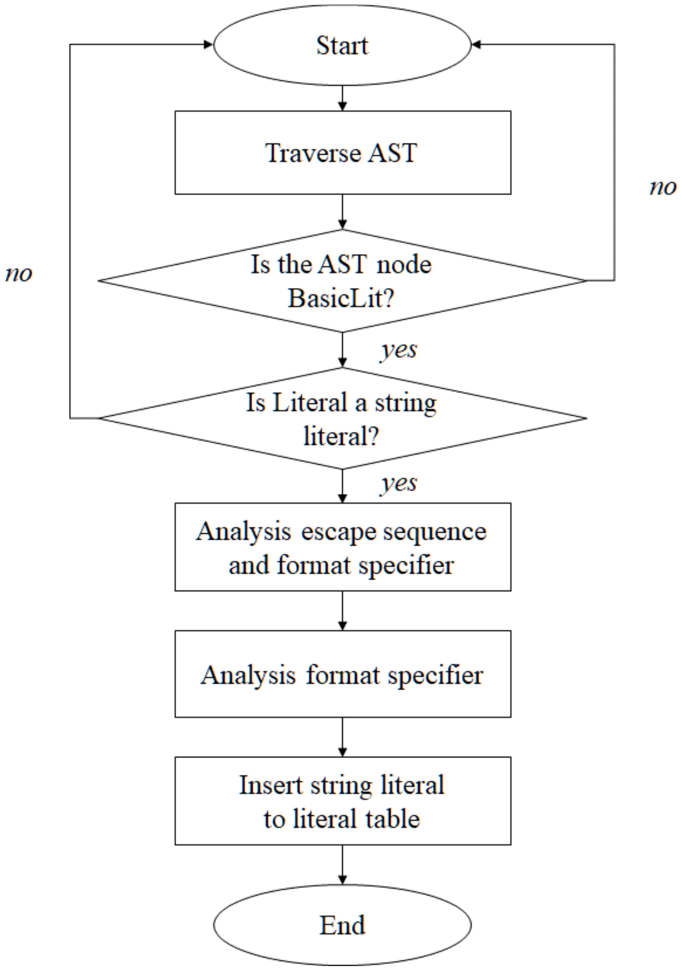

Start
Traverse AST
Is the AST node BasicLit?
no
no
yes
Is Literal a string literal?
yes
Analysis escape sequence and format specifier
Analysis format specifier
Insert string literal to literal table
End

Figure 2. Literal Table Generation Process

Symbol Table Generator

The symbol table generator generates a table that stores information about symbols needed to convert the Golang source code to intermediate code. Figure 3 shows the relationship between the symbol table and another tables. The symbol table generator, while traversing

AST, produces a total of five tables: symbol table, abstract table, member table, function table, and access to each table via the mind of the symbol table. A symbol table is a table that stores basic information about variables and function symbols with global/local scopes, such as type, symbol name, and offset.

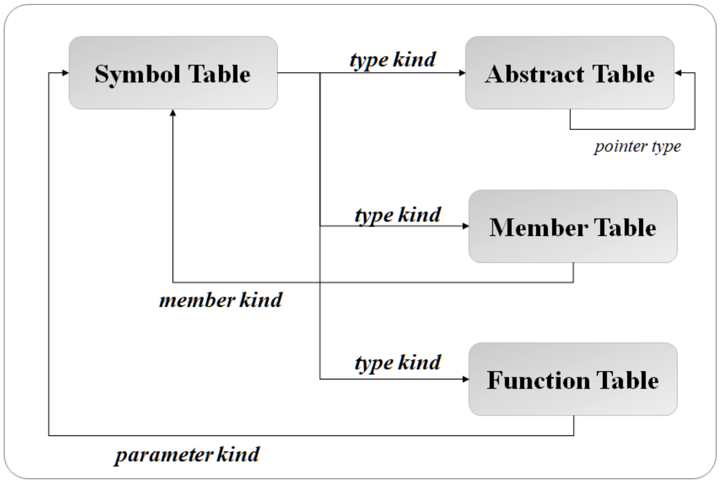
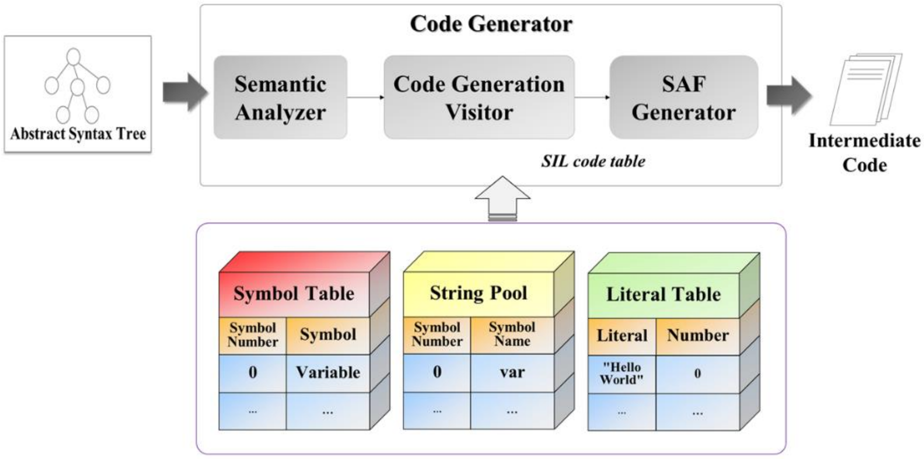


Figure 3. Relationship between the symbol table and another tables

An abstract table is a table that stores symbolic information such as pointers, arrays, maps, etc., that cannot have substantial storage space. The abstract table manages the type of pointing target if the symbol is a pointer, or the type of information stored in the map or array. The member table stores member variables and symbol information of the member function of the structure and stores member kinds that access the symbol table to obtain information such as the member structure identifier, member identifier, and type and offset for the member. The function table stores function symbol information, the parameter identifier list of the function, the return type list, the named return type list, and the receiver identifier.

Intermediate Code Generator

The intermediate code generator traverses the abstract syntax tree and converts the source code into an intermediate code format or the smart intermediate language (SIL). The intermediate code generator in this paper consists of a node-type inference machine, code generation non-visor, and the smart assembly format (SAF) generator, which converts a source code into SIL using information from information tables such as symbol tables, string pools, and literal tables. Figure 4 shows a intermediate code generator structural diagram.



Symbol Table	
Symbol Number	Symbol
0	Variable
...	...

String Pool	
Symbol Number	Symbol Name
0	var
...	...

Literal Table	
Literal	Number
"Hello World"	0
...	...

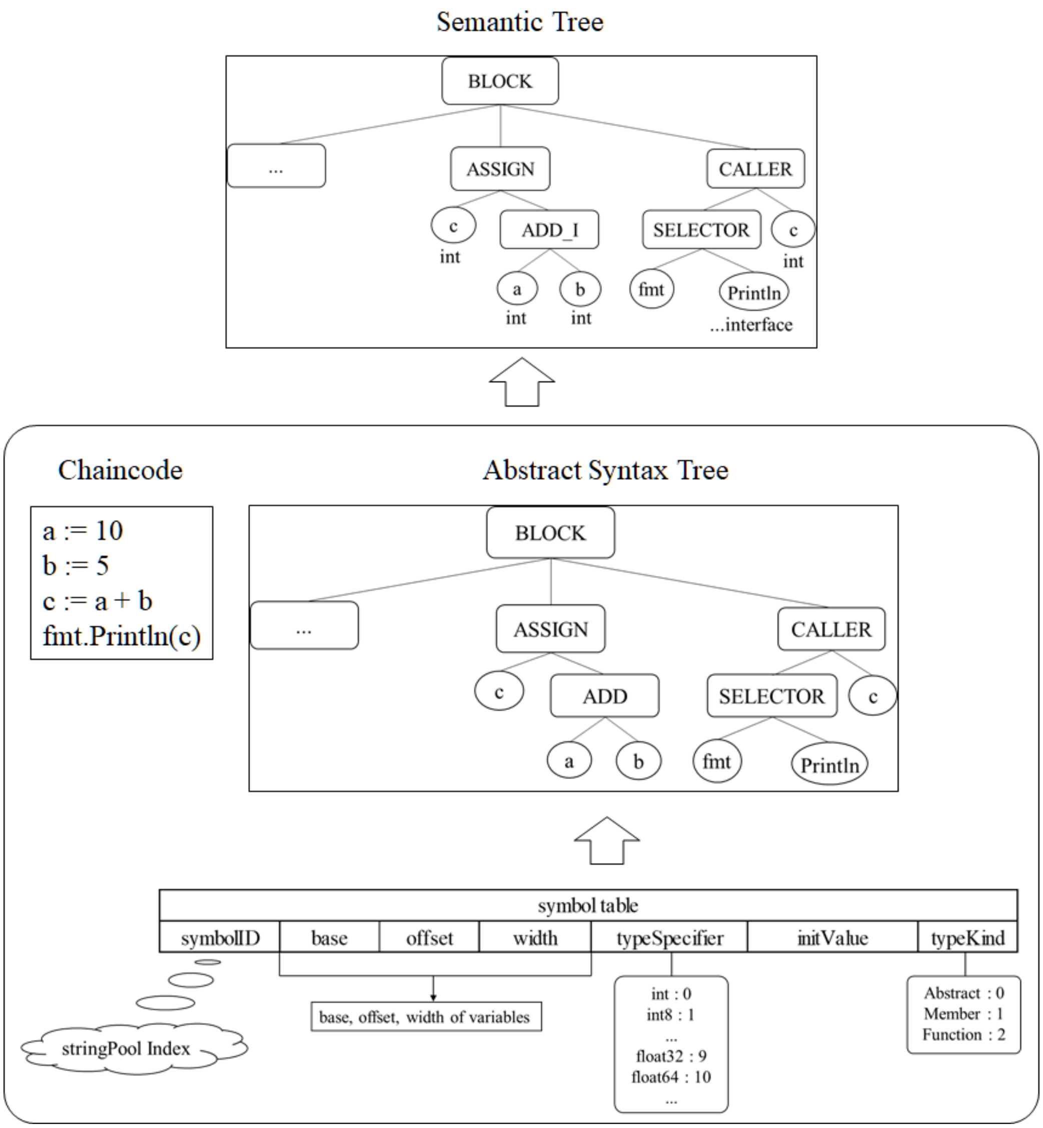
Figure 4. Intermediate Code Generator Model

Semantics Analyzer

The AST has only the meaningful information needed to generate intermediate code by removing unnecessary nodes from the past tree. However, code generation using AST alone cannot be checked for correct type schemes, which can result in semantic defects in generated code.

The semantic analyzer uses the symbol table and Golang's type system to explore the AST and analyze the presence of a semantic defect. The process is as follows.
1. If you visit the terminal node of the node during the AST traversal, specify the type of terminal node information of the symbol table. 2. Visiting a non-terminal node during the AST tour analyzes the existence of semantic defects in the non-terminal node according to the type of the lower node and type system, and specifies the node type to facilitate code generation.
Figure 5 shows an example of semantic analysis.

In Figure 5, the type information of a,b, the terminal node of AST, is accessed and imported from the symbolic table and the type of terminal node is specified. The ADD nodes, which are non-terminal nodes, verify that terminal nodes a and b are of the same type according to the Golang type system, determine that the two types are semantically non-defective, and combine type information in favor of code generation. Subsequently, the non-terminal node ASSIGN verifies that the terminal node c and the non-terminal ADD node are of the same type according to the type system and determines that there is no semantic defect if the type is the same.



```
a := 10
b := 5
c := a + b
fmt.Println(c)
```

symbol table						
symbolID	base	offset	width	typeSpecifier	initValue	typeKind

Figure 5. Example of Semantic Analysis

For non-terminal nodes related to function calls, the parameter type of the function is taken from the symbol table and assigned to the terminal node Println. The non-terminal node SELECTOR is assigned an interface, a type of function parameter. Finally, the non-terminal node CALLER checks whether the parameter type of the SELECTOR node includes the type of terminal node c.

Code Generation Visitor

Code generation non-visitors generate intermediate code while traveling through semantic trees generated by semantic analysts. The code generation process is done through pattern matching, and is largely divided into expression and statement processing.

Expression processing is largely divided into a symbol processing and a operator processing. The symbol processing uses information from the information table to generate intermediate code using the base, offset, address, and value information of stored symbols. Operator processing uses the type information given to the semantic tree to determine the type of operator, and generates intermediate code using a code matrix that maps this type information to intermediate code.

Sentence processing produces sentences such as conditional statements, repetitive statements, and assigned statements in semantically equivalent intermediate codes. Each sentence generates an intermediate code to match the intermediate code pattern of the sentence. Figure 6 shows the intermediate code pattern of the repeat statement.

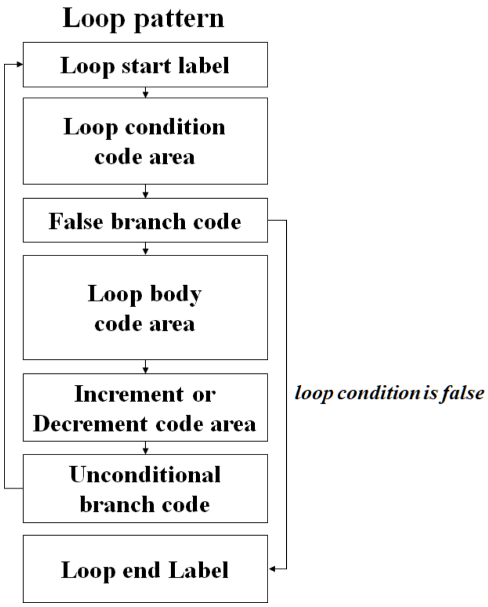


Figure 6. Intermediate Code Pattern of the Repeat Statement

In the intermediate code SIL, the iteration pattern begins with a start label to signal the beginning of the iteration. Then, an intermediate code region appears for the iteration condition and a branch code determining the iteration termination based on the iteration condition. The body code area of the repeat statement that runs when the repeat condition is satisfied appears, as well as the code area that updates the initial expression of the repeat condition appears. Finally, after the execution of the repeat body is completed, an unconditional branch code that branches to the starting point of the repeat statement, followed by and an end label that indicates the end of the repeat statement.

Figure 7 illustrates the code generation process of repetitive statements through pattern matching. Code generation visors are generated by matching repeat patterns when generating intermediate code for repeat statements. First, a repeat start label is generated, and if a condition node in the semantic tree exists, a code that corresponds to a condition statement or a condition code for an infinite loop will be generated. The loop termination branch code is then generated followed by the recurrent body code. Finally, if an initial value update node exists, it generates an increasing or decreasing expression; otherwise, it generates a code that branches unconditionally to the starting point of the iteration, and then generates a repeat end label.

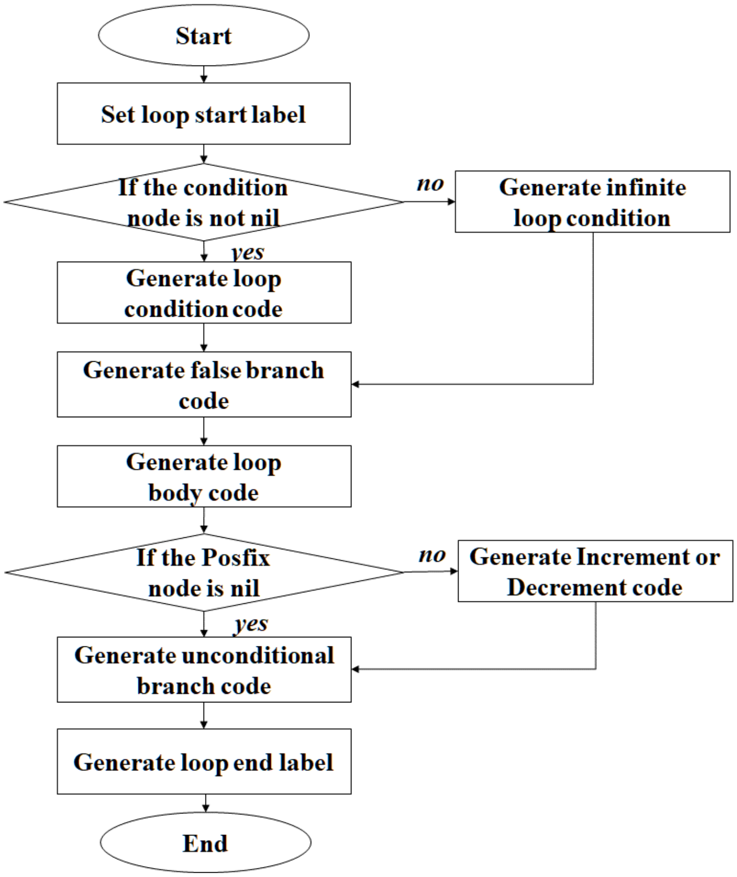

Start
Set loop start label
If the condition node is not nil
no
Generate infinite loop condition
yes
Generate loop condition code
Generate false branch code
Generate loop body code
If the Posfix node is nil
no
Generate Increment or Decrement code
yes
Generate unconditional branch code
Generate loop end label
End

Figure 7. Code Generation Process of the Repeat Statement

SAF Generator

The Smart Assembly Format (SAF) generator generates assembly files using intermediate code tables and information tables generated by the code generation visor. Figure 8 shows a structure of the SAF generator.

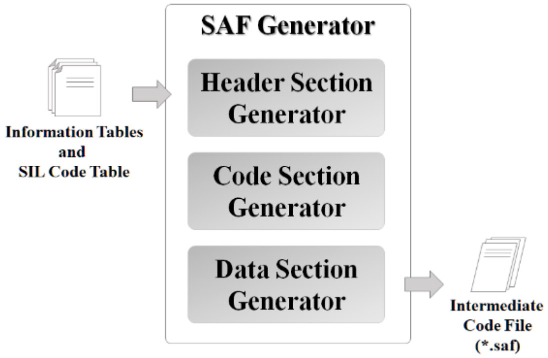


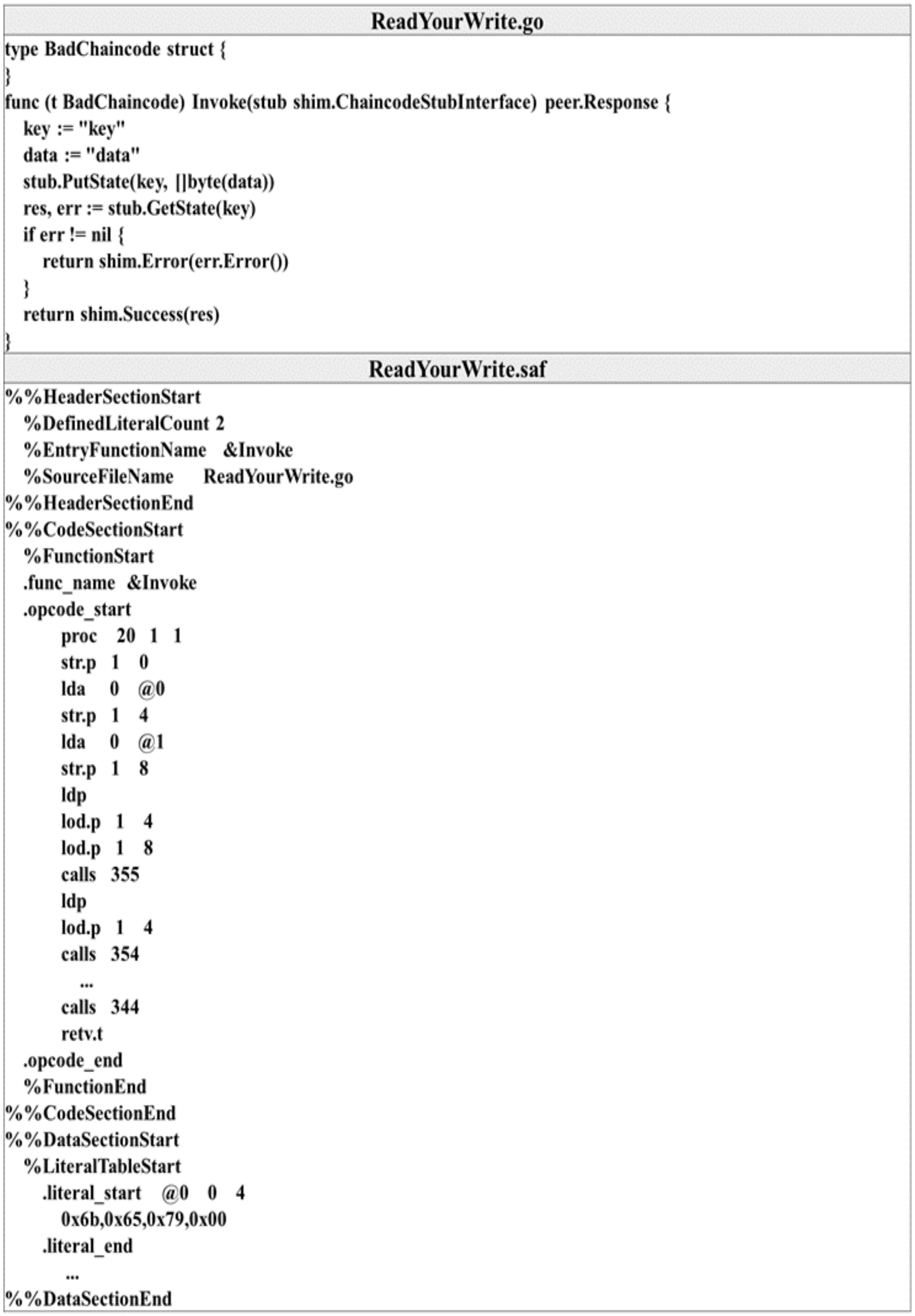
Figure 8. Structure of the SAF Generator

The SAF generator consists of a header section generator, a code section generator, and a data section generator. The header section generator generates information such as the number of literals defined, the number of initialized variables, the number of uninitialized variables, and the name of the entry function, representing the key information that constitutes the SAF file. When creating a header section, the generator uses the information from the literal table and the symbol table to generate each information in a format.

The code section generator generates SIL code information for the function, which is the information that represents the execution code of the program. When generating code sections, the SAF generator uses SIL code table information to generate them in a format. The data section generator generates global variables and literal information referenced in the code domain, the information that represents the data in the program. The generator takes global variable information from the symbolic table and generates it according to the format, and converts literals into hexadecimal numbers using literal table information to generate data section information.

Experimental Results and Analysis

To ensure that the intermediate code generator for the proposed chaincode security weakness analysis runs normally, the chaincode embedded in the security weakness is converted to intermediate code. The experiment was conducted using two examples: ReadYourWrite.go and UnhandledError.go. Figures 9 and 10 show the results of the intermediate code conversion of each example.



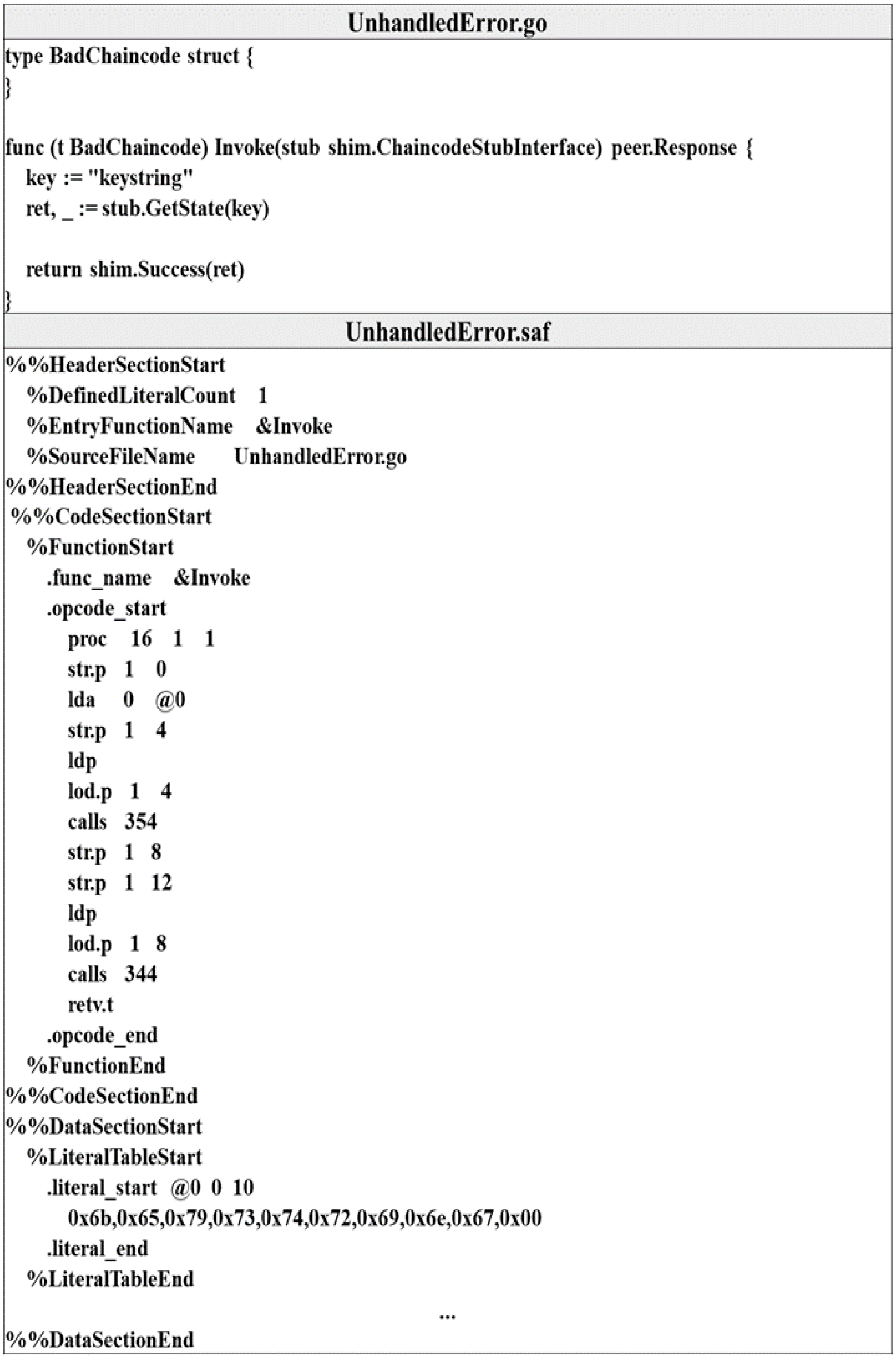
ReadYourWrite.go

```
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)
}
```

ReadYourWrite.saf

```
%%HeaderSectionStart
  %DefinedLiteralCount 2
  %EntryFunctionName   &Invoke
  %SourceFileName      ReadYourWrite.go
%%HeaderSectionEnd
%%CodeSectionStart
  %FunctionStart
  .func_name  &Invoke
  .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
          ...
        calls  344
        retv.t
  .opcode_end
  %FunctionEnd
%%CodeSectionEnd
%%DataSectionStart
  %LiteralTableStart
    .literal_start   @0   0   4
        0x6b,0x65,0x79,0x00
    .literal_end
        ...
%%DataSectionEnd
```

Figure 9. Intermediate Code Generation Results for ReadYourWrite.go



UnhandledError.go

```
type BadChaincode struct {
}

func (t BadChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
   key := "keystring"
   ret, _ := stub.GetState(key)

   return shim.Success(ret)
}
```

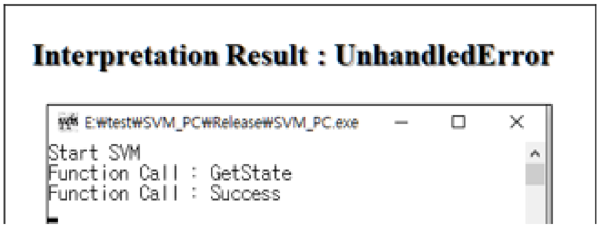
UnhandledError.saf

```
%%HeaderSectionStart
   %DefinedLiteralCount    1
   %EntryFunctionName    &Invoke
   %SourceFileName      UnhandledError.go
%%HeaderSectionEnd
 %%CodeSectionStart
   %FunctionStart
      .func_name    &Invoke
      .opcode_start
         proc    16   1    1
         str.p   1    0
         lda     0    @0
         str.p   1    4
         ldp
         lod.p   1    4
         calls   354
         str.p   1   8
         str.p   1   12
         ldp
         lod.p   1   8
         calls   344
         retv.t
      .opcode_end
   %FunctionEnd
%%CodeSectionEnd
%%DataSectionStart
   %LiteralTableStart
      .literal_start  @0  0  10
         0x6b,0x65,0x79,0x73,0x74,0x72,0x69,0x6e,0x67,0x00
      .literal_end
   %LiteralTableEnd

                                         ...
%%DataSectionEnd
```

Figure 10. Intermediate Code Generation Results for UnhandledError.go

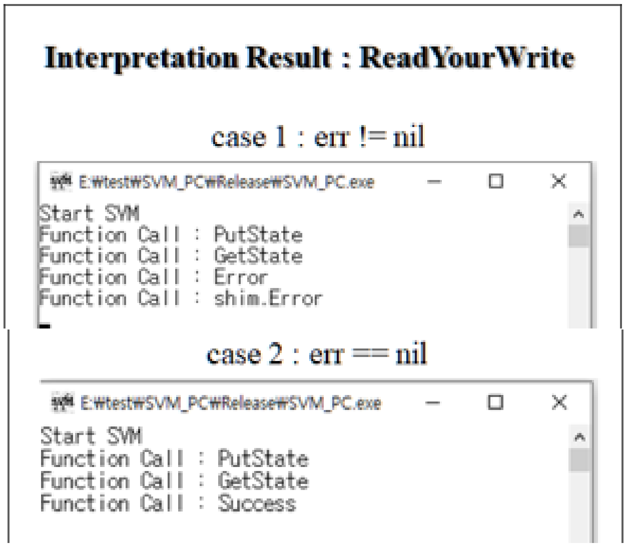
Next, the generated intermediate codes were run as a virtual machine, namely the Smart Virtual Machine (SVM) (Cousot, Cousot, 2002), to verify that the intermediate code generation of the chaincodes was not defective. Figures 11 and 12 show the results of running the generated intermediate code as a virtual machine SVM.



Interpretation Result : UnhandledError

```
Start SVM
Function Call : GetState
Function Call : Success
```

Figure 11. Execution Results of UnhandledError.go Program


E:₩test₩SVM_PC₩Release₩SVM_PC.exe
E:₩test₩SVM_PC₩Release₩SVM_PC.exe

Interpretation Result : ReadYourWrite

case 1 : err != nil

```
Start SVM
Function Call : PutState
Function Call : GetState
Function Call : Error
Function Call : shim.Error
```

case 2 : err == nil

```
Start SVM
Function Call : PutState
Function Call : GetState
Function Call : Success
```

Figure 12. Execution Results of ReadYourWrite.go Program

Conclusion and Future Research

The smart context, which operates in the hyperledger fabric framework, is written by implementing an interface called a chaincode. When implementing a chaincode, there may be a security weakness inside the code, which is the root cause of the security vulnerability. However, when the contract is completed and the block is created, the chaincode cannot be arbitrarily modified, so the security weakness must be analyzed before execution.

This paper conducted a study on chaincode intermediate code generators to analyze the security weaknesses of a chaincode operating in the hyperledger fabric framework. Security weakness analysis at the source code level is not easy to apply security weakness analysis techniques because the code logic is not clear and complex. The intermediate code generated by the intermediate code generator clearly represents the code logic of the source code, is less complex than the source code, so it is easy to analyze all the execution paths of the program. It is also suitable for dynamic analysis because it can be executed by intermediate code interpreters.

Currently, the intermediate code generated by the intermediate code generator has confirmed that the code runs in the virtual machine SVM without any problems. In the future, we will study the security weakness dynamic analyzer of chaincodes using SVM, and the security weakness static analyzer that analyzes intermediate codes using static analysis techniques such as data flow analysis, symbolic execution, and control flow analysis.

Acknowledgements

This research was supported by Seokyeong University in 2021.

References

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

Aho, Alfred V., Sethi, Ravi, Ullman, Jeffrey D. (1986). Compilers Principles, Techniques and Tools, Addison-Wesley.

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

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

Cousot, P., Cousot, R. (2002). Modular Static Program Analysis, *LNCS*, 2304, 159-178.

Fagan, M. E. (1976). Design and Code Inspections to Reduce Errors in Program Development, *IBM Systems J.*, 15(3), 182-211.

https://www.hyperledger.org/

Jeong, J., Son, Y., Lee, Y. (2019). A Study on the Secure Coding Rules for Developing Secure Smart Contract on Ethereum Environments, *International Journal of Advanced Science and Technology*, 12, 47-58.

Kim, S., Son, Y., Lee, Y. (2020). A Study on Chaincode Security Weakness Detector in Hyperledger Fabric Blockchain Framework for IT Development, *Journal of Green Engineering*, Alpha Publishers, 10(10), 7820-7844.

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.

Son, Y., Lee, Y. (2012). A Study on the Smart Virtual Machine for Executing Virtual Machine Codes on Smart Platforms, *International Journal of Smart Home*, SERSC, 6(4), 93-105.

Son, Y., Lee, Y. (2014). Smart Virtual Machine Code based Compilers for Supporting Multi Programming Languages in Smart Cross Platform, *International Journal of Software Engineering and Its Applications*, SERSC, 8(5), 249-260.

Son, Y., Jeong, J., Lee, Y. (2020). Design and Iimplementation of an IoT-Cloud Converged Virtual Machine System, *The Journal of Supercomputing*, Springer, 76(5), 5259-5275.

Son, Y., Lee, Y., Oh, S. (2015). A Software Weakness Analysis Methods for the Secured Software, *The Asian International Journal of Life Sciences*, 12, 423-434.

Wichmann, B.A., Canning, A.A., Clutterbuck, D.L., Winsbarrow, L.A., Ward, N.J. Marsh, D.W.R. (1995). Industrial Perspective on Static Analysis, *Software Engineering Journal*, 10(2), 69–75.