International Journal of Multimedia and Ubiquitous Engineering
Vol.12, No.10 (2017), pp.15-26
http://dx.doi.org/10.14257/ijmue.2017.12.10.02

Intermediate Language Translation and Evaluation for Binary Code Software Weakness Analysis

Junho Jeong[1], Yunsik Son[2] and Seman Oh[2*]

[1]*Electronic Commerce Institute, Dongguk University*
[2]*Dept. of Computer Science and Engineering, Dongguk University*
{yanyenli, sonbug, smoh}@dongguk.edu

Abstract

The use of third-party libraries has become a natural phenomenon as more programs are built on a large scale. The problem with using third-party libraries is that, in most cases, we only have access to the binary code. And due to the lack of access to source code, software weakness analysis is very difficult. In this paper, we propose a method to translate binary code to Smart Intermediate Language (SIL) for software weakness analysis.

Keywords*: Software weakness Analysis, IL Translation, Binary Code*

1. Introduction

Recent uses of software have moved beyond simple applications to environments such as military weapons, medical care, and automobiles. The problem with this is that a variety of security incidents occur due to vulnerabilities inherent in such software, which can lead not only to monetary loss, but also to loss of life [1-4].

In order to eliminate the vulnerability in software, various studies have been carried out to remove software weaknesses that could lead to vulnerabilities. Among these studies, research on static analysis based on source code to define, analyze and remove software weaknesses that can be a threat in the development stage.

However, as computer software is manufacture on a large scale, it is natural to use libraries that have been created by a large number of people as well as third-party libraries. And, third-party libraries mare generally provided in the form of binary code rather than source code [5, 6]. It is difficult to analyze and remove software weaknesses in the development stage without the source code. This is because the binary code removes a large amount of program syntax and semantic information that exists in source code.

This study was conducted to analyze the various software weaknesses in binary code. The initial research was carried out through reverse engineering, but the process became complicated and the analytical process was different and the efficiency was poor [7-9]. To overcome this, a method of analyzing software weaknesses using an intermediate language has been proposed [10-13]. This method has the advantage of being able to utilize the same analysis method regardless of the type of binary code. However, the problem of designing an intermediate language that is effective in analyzing software weaknesses and converting missing information such as variables, data types, and function information from binary code to intermediate language.

In this paper, we propose an effective intermediate language for software weakness analysis and analyze the translation from binary code to intermediate language to analyze software weaknesses using intermediate language. In Section 2, we discuss major software weaknesses and their analysis methods based on binary code, and introduce the

Received (August 6, 2017), Review Result (October 7, 2017), Accepted (October 11, 2017)
* Corresponding Author

ISSN: 1975-0080 IJMUE

intermediate language and translation techniques in Section 3. Section 4 analyzes the results of the transformation. Our conclusions and future works of study are discussed in Section 5.

2. Static Software Weakness Analysis on Binary Code

ITS4 was the earliest proposal to solve software security problems through source [14]. Prior to the introduction of static analysis tools to analyze security problems using source code, errors and bugs in the source code were corrected by feedback through the error reported by the compiler. However, vulnerabilities that are not easily found have become increasingly dangerous over time, and the cost of correcting them have also increased, requiring more powerful static analysis in addition to compilers.

Theoretically, examining source code is equivalent to examining compiled programs, but checking individual source code is a time-consuming task, and although effective, code inspectors must be aware of security vulnerabilities. However, with static analysis tools, you can perform repetitive code reviews more quickly and with the equivalent security knowledge to that of the security inspector. Static analysis methods can be tested more simply than dynamic analysis, and must be performed at a level of meaningful compilation.

Among the software weaknesses that can be detected using static analysis, Memory Corruption causes security problems such as changing program execution flow, elevation of privilege and by-passing authentication. Examples of specific attacks using such memory contamination include buffer overflow, use after free (UAF), and integer variable overflow. Memory corruption often occurs in C / C ++, where complex memory structures can be used.

There are various tools to analyze software weaknesses in source code. However, there is no perfect tool or analysis method to analyze software weaknesses of binary code. This study analyzes binary code as follows.

2.1. Buffer Overflow Weakness

Stack and Heap buffer overflows were the most common security problems in the mid and late 2000s have continuously been reported. The security problem is caused by the presence of a vulnerable function in the source code that does not check the size of the buffer and executes the user's command. Especially when data input from external sources (files, sockets, etc.) are used for these vulnerable functions, which can cause security problems.

Rawat et al. have detected a buffer overflow loop (BOIL) in the binary file [15]. If such a BOIL exists, it is defined as a buffer overflow vulnerable function (BOP) and proposed based on the BinNavi reverse engineering framework. The initial binary executable code is converted into assembly code through IDA Pro and converted into intermediate language through REIL. After that, we analyze the buffer overflow by creating a module (Static Analysis Module) that can be accessed through the Jython API through the Call Graph and Control flow graph (CFG) provided by REIL.

Lee et al. proposed a tool called BinaryReviser to detect buffer overflows in binary files [16]. The tool directly analyzes the binary file to find the part that causes a buffer overflow. It also able to remove vulnerabilities through code patching.

2.2 Use After Free Weakness

Use After Free(UAF) software weakness is the most common software weakness since the late 2000s. According to Common Vulnerabilities and Exposures(CVE) reports, more than 60% of recent security problems are caused by use after memory release.

A typical example of this analysis is that Feist and others convert binary files into REIL intermediate languages to find UAF software weaknesses. In this study, we propose

GUEB (Graph of Use after Exploit Binary) tool and define UAF software weakness through the UAF sample source code. The vulnerability was actually found in the ProFTPD program and its contents are registered as CVE 2011-4130. However, loop statements in source code are only executed one, ignored, and the defined content are not formalized. And, the integrated environment cannot be provided.

2.3 Integer Overflow Weakness

Integer overflows are not directly used for attacks, but they cause problems such as denial of service attacks or program malfunctions. Integer variables are often used to determine execution paths in branch statements (if, etc.) or loop statements (for, while, etc.), and it is difficult to predict the program execution flow if integer variable overflow occurs. Corner Case) occurs.

IntScope proposed by Wang et al. Proposed a method to detect a phenomenon called overflow, which is common in variable types storing integer values such as Integer type whose memory size is fixed [17]. It converts the binary executable into an intermediate language called PANDA, and in the process, creates a CFG and call graph. Then, using the component extractor and the profile constructor, we can select the flows that are directly or indirectly connected to the sinks function and creates a separate chop function. The generated chop function is used to output a flow suspected to be an integer type variable overflow using a depth-first search method. However, the final integer variable overflow must be determined by the user.

3. Binary Code to Intermediate Language Translation

In this paper, we propose a system for selecting an effective intermediate language to analyze the software weakness inherent in the binary code, and converting the binary code to the corresponding intermediate language. To verify the conversion result efficiently, we analyze the control flow for the binary code and intermediate language code and implement the tool to visualize the result.

3.1 Smart Intermediate Language for Software weakness

Based on previous studies, there are three major factors to effectively analyzing major software weaknesses. First, it should be able to fully express information about variables and their data types. It is very difficult to infer variables and data types from binary code or assume that such information is sufficient to be represented in the intermediate language. Secondly, the control flow of the program should be sufficiently expressible and the control flow unit should be easy to analyze. This is because it is a major software weakness where the branch from the control flow to the wrong code occurs. Third, it should be able to express information about the function. The use of vulnerable functions is one of the most significant software weaknesses. It is important to be able to perform not only predefined vulnerable functions but also user defined functions based on information on return types and parameters for functions used in the program. For this reason, we selected the SIL as an intermediate language for analyzing software weaknesses.

The SIL is a language designed for operation in stack-based virtual machines, independent of the programming language and hardware platform. It is divided into 7 operation codes as shown in figure 1. Since the operation code of the language is basically stack-based code, the operand used in the operation is taken from the operand stack and the result of the operation is stored in the operand stack. It also has a maximum of two operands, and has a single result through an instruction. It is structured as <operand 1, operand 2, operation result>. An opcode is represented by two bytes and can have instruction parameters as needed. The enumeration value of the opcode processed by the interpreter and the mnemonic to be expressed in the SAF text format are defined. The

mnemonic of the operation code is a language defined by a combination of alphabets and integers meaning operation for readability of code. When type information is needed according to the type of operation code, the type symbol is padded with the symbol '.'(dot). Therefore, each operation satisfies the first condition including the data type information of the operand performing the operation, the control flow is sufficiently expressed, and the control flow unit can be effectively analyzed. In addition, existing functions can be predefined, and user-defined functions can be effectively expressed, which can be effectively utilized for software weakness analysis.

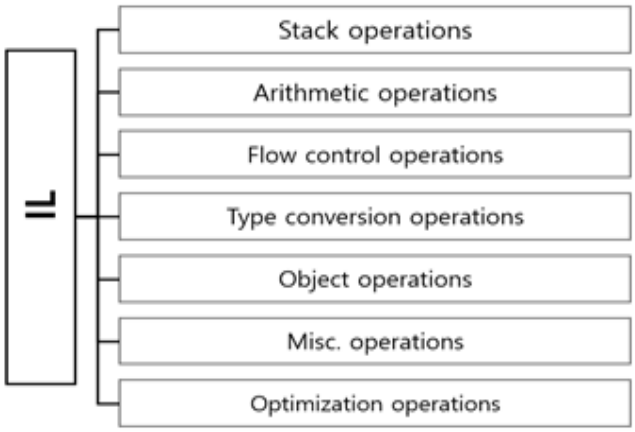

Figure 1. Proposed Language Opcode Categories

3.2 Binary Code Analysis and Intermediate Language Translation

There were some problems with the translation into the SIL presented by analyzing the binary code. The first is that there are too many assembly codes. Converting all of these assembly code to SIL is a very inefficient task. Even assembly instructions are CISC (Complex Instruction Set Computer) instructions, and SIL is a Reduced Instruction Set Computer (RISC) instruction. Therefore, to overcome this problem, we replaced the assembly code with the RISC instruction in the assembly instruction and converted it into the corresponding SIL.

Table 1 shows the selected generalized set of assembly instructions. We have solved this problem by allocating independent stack space to store EFLAGS and register information, and implemented RISC instructions that do not use flags and conditional branch instruction.

Another problem is that x86 / 64 commands are basically pipeline structure commands. Each pipeline has different registers, but if the pipeline is shortened on compilation, it becomes a problem. However, it is not a factor when solving the first problem, because the conversion is rather long.

Table 1. Selected General X86 Instruction Set for SIL Translation

Stack management	nop	pop	push	
Logical & arithmetic operations	mov	add	sub	xor
	or	and	xchg	shl
	sal	shr	sar	rol

	ror	not	mul	imul
	div	idiv	neg	inc
	dec	cmp	test	
Control flow	jmp	ja	jnbe	jae
	jnb	jb	jnae	jbe
	jna	jg	jnle	jge
	jnl	jl	jnge	jle
	jge	jz	jnz	ret
	call			
String processing command	rep	movs	lods	stos

However, the problem remains that one assembly instruction is replaced by too many SIL instructions. Nonetheless, the lengthy sentence itself does not compromise the syntax and semantics of existing code, so it is not require converting from binary code to SIL. The process of converting the binary code into the SIL in the proposed system can be expressed in three stages. The first step is to analyze binary files written in C / C ++ and classify them by analyzing the written language and type of files. Even with the same source code, different types of binary code are generated depending on the compiler and the target machine, so pre-classification is required. Therefore, the proposed system performs analysis on x86 / 64-bit binary code in PE format written in C / C ++, and treats other objects as errors.

The second step is to analyze the binary code to extract each function used in the program and convert the extracted functions into assembly code by constructing a general block based on the CFG.

Figure 2 shows two memory concepts RAW and RVA (Related Virtual Address) used in the PE format structure as input of the proposed system. RAW is the offset used in the file. It is called RVA when the actual program is reallocated when it is loaded into memory. In consideration of this, it is necessary to correctly calculate the inverse of the function to be extracted so that it can be determined from which position of the file to extract.

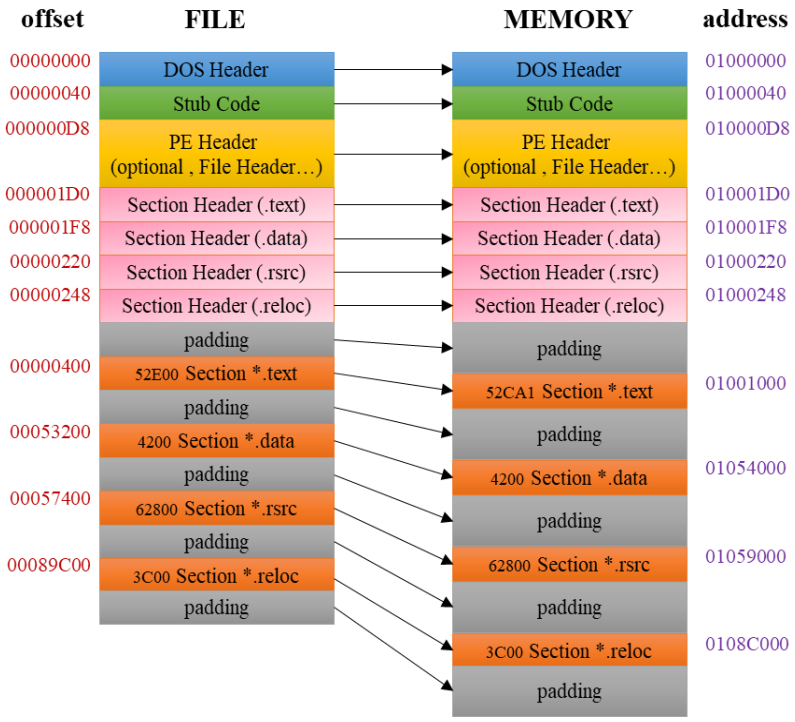

Figure 2. PE Structure (RAW, RVA)

The main section analyzed in the binary code is the .rdata area, and the .rdata area contains the IMAGE_EXPORT_DIRECTORY structure, which contains information about how many functions that code exports, what the name is, and what offset (RVA). We could use this information to distinguish and extract functions that exist in the binary code.

CFG is important because it can analyze the control flow of the program, understand the overall program configuration, and scan the basic blocks to build programs more efficiently. The proposed system does not show the calling relation of each function but the flow is separated and expressed as CFG when a branch according to the condition appears.

The final step is to convert the assemblies into generalized assembly instructions of our choosing and convert them into SIL that contain information on data types based on CFG and basic data type inference.

In this process, x86 / 64 has a register in the CPU, but the SIL to be converted has a one-to-one correspondence between the register and the specific area of the stack so that it could act as a register. That is, the register is also considered to be a kind of stack succession and is converted. Therefore, the role of registers in the existing SIL as shown in <Table 2>, has been previously allocated to the stack area.

In addition, SIL requires data type deduction for variables from binary files because the instructions vary depending on the data being handled. In the proposed system, we find the pattern [ebp ± constant], [esp ± constant], [ebp + register * constant-constant] patterns and set char, short, int variable and int array types.

Table 2. Mapping of Registers and Stack Addresses

Register Name(x86)	stack (base, offset)
rax	(0, 0)
rbx	(0, 8)
rcx	(0, 16)
rdx	(0, 24)
rsp	(0, 32)
rbp	(0, 40)
rsi	(0, 48)
rdi	(0, 56)
rip	(0, 64)

4. Experiments

To analyze the performance of the converter, we generated the source code with software weakness as binary code, parsed the corresponding binary code into assembly code, parsed the function and generated CFG accordingly. This is a very important step in code conversion and performance analysis. Because the source code and the following process must be converted correctly from the first binary code to the intermediate language without losing the semantics of the existing code, it is very inefficient to simply analyze it with the code. To overcome this, we implemented a visualization tool that can effectively analyze the CFG to effectively analyze and evaluate the results of binary code to SIL conversion.

The process of generating CFG at the time of code conversion is as follows. Parsing the binary code extracts the functions of the assembly type, and analyzes and graphs the control flow based on the parsed file. The basic blocks composing the CFG are constructed based on the branch, and the control flow is expressed through the trunk line between the basic blocks. So, we implemented the visualization tool using Dagre open source which can take the JSON format as an input and express blocks and trunks. Then, CFG of binary code and SIL code is generated using this tool, and the performance of the conversion technique is analyzed by comparing and analyzing the results. Figure 3 below shows the JSON format for CFG representation. The contents of the basic block are expressed as "block name [label =" text content "]" and the edges connecting the blocks are expressed as "block name -> block name".

```
digraph {
   BlockName[label="BlockContents"];
   BlockName->BlockName
}
```

Figure 3. JSON Format for CFG Representation

Since we created the binary from the source code, we know the CFG of the program and compare the results of the CFG generated from the assembly code. In this study, we tested transformations using various test cases. Figure 4 below shows the source code where Integer Overflow software weakness is embedded. When the wrapperMalloc function is called in the integerOverFlowExam function, the second parameter, i, is used as a parameter to the mal-loc function. The value of i may be negative. The parameter of

the malloc function in C language is size_t, which represents only a positive number, so the source code performs very large dynamic allocation when i is input as a negative number.

```
void wrapperMalloc(void** temp, int i) {
   (*temp) = (int*)malloc(sizeof(int)*i);
}
void integerOverFlowExam( ) {
   int i, j;
   int *ap = NULL;
   void* ptr;
   scanf("%d %d", &i, &j);
   wrapperMalloc(&ptr, i);

   ap = (int*)ptr;

   for (int k = 0; k < i; k++) {
      if(k > j) {
         *(ap + k) = 0 ;
      } else {
         *(ap + k) = k + 1;
      }
      printf("%d", *(ap + j));
   }
}
```

Figure 4. An Example Program for Analysis

```
digraph {

2656 [label="2656
00000000 55 push ebp
00000001 8b ec mov ebp, esp
00000003 81 ec c0 00 00 00 sub esp, 0xc0
00000009 53 push ebx
0000000a 56 push esi
0000000b 57 push edi
0000000c 8d bd 40 ff ff ff lea edi, [ebp-0xc0]
00000012 b9 30 00 00 00 mov ecx, 0x30
00000017 b8 cc cc cc cc mov eax, 0xcccccccc
…
"];

2768 [label="…"];
2857 [label="…"];
2866 [label="…"];
2874 [label="…"];
2882 [label="…"];
2897 [label="…"];
2912 [label="…"];
2914 [label="…"];

2768 -> 2866
2857 -> 2866
2866 -> 2874
2866 -> 2914
2874 -> 2882
2874 -> 2897
2882 -> 2912
2897 -> 2912
2912 -> 2857
}
```

Figure 5. JSON Format Data for CFG Representation of Example Code

The assembly code generated from this source code and the JSON formatted data for the CFG representation of the code are shown in Figure 5. the block name and contents of

the corresponding block are described in Figure 3. Finally, the CFG is expressed by representing the flow trunk of the blocks. To analyze whether this data is equivalent to the control flow of actual source code, we can obtain the visualized CFG result as shown in Figure 5 using our own visual tool.

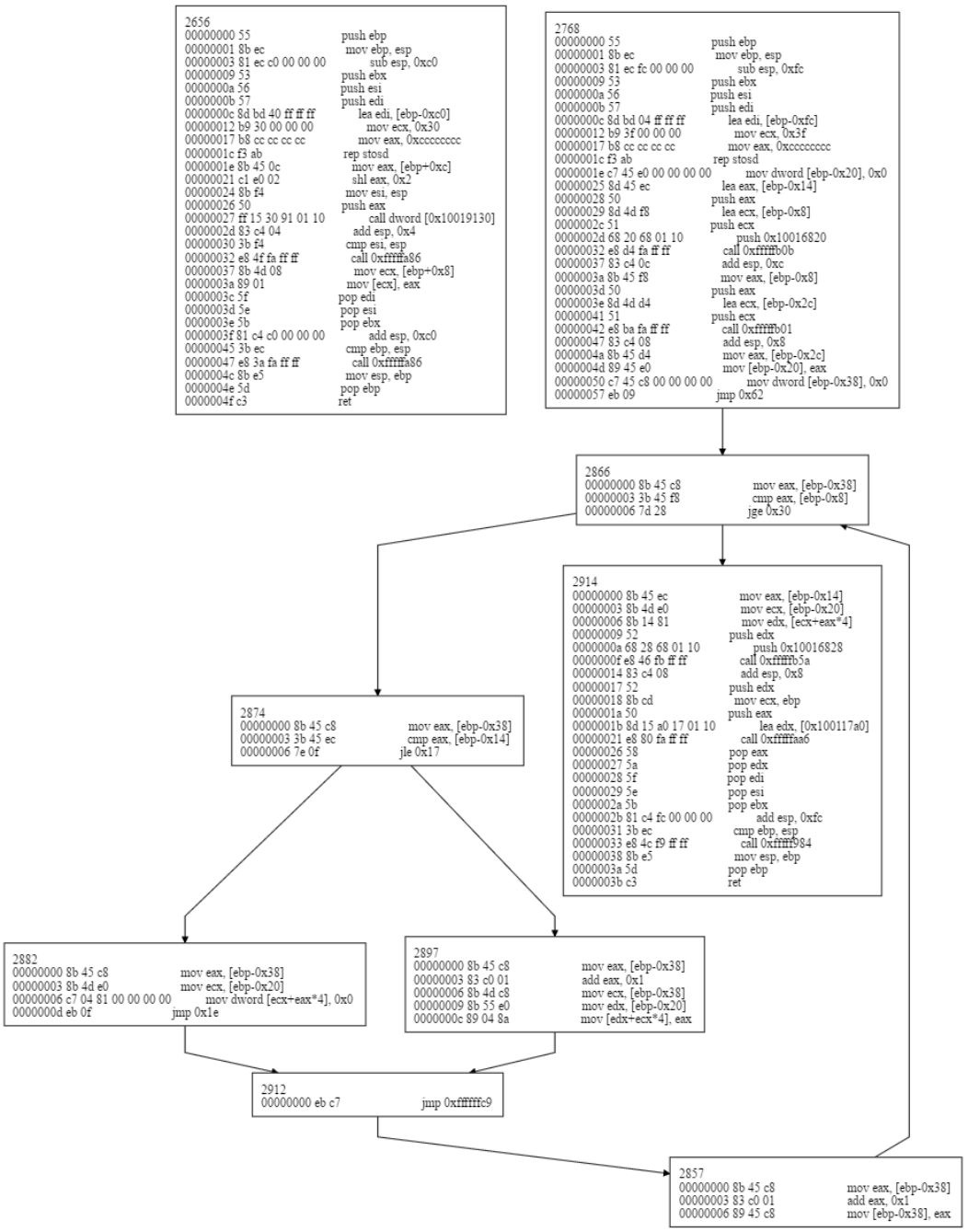

Figure 6. The CFG Generated from the Binary Code

In addition, SIL requires data type deduction for variables from binary files because the instruc-tions vary depending on the data being handled. In the proposed system, we find the pattern [ebp ± constant], [esp ± constant], [ebp + register * constant-constant] patterns and set char, short, int vari-able and int array types. scanf is called first in block 2768, and int type variables i and j are input. i is used as the size of the dynamically allocated array, and j is set in the local variable to be used as a position to set the value. Finally, the start

address of the dynamically allocated array is allocated to the [ebp-0x20] address corresponding to the ap variable, and the block is finalized after initialization of the loop statement.

In block 2866 k and i are compared, if k is smaller, it is compared with j in block 2874. If k is still smaller, we go to block 2897, insert a value into the array corresponding to k index, and increment k by 1 in block 2857. If k is smaller than i but larger than j, go to block 2882, put 0 in the corresponding array, go to block 2857, and increment k by 1. Finally, if k is not less than i, the call to a function prints the value corresponding to the index of j in array # 2914.

As mentioned earlier, the software weakness of the source is that a negative number can be entered when calling wrapperMalloc, and the information necessary for analyzing the flow and the function can be seen in Figure 6. Therefore, if the control flow of the corresponding code and execution contents are well converted to the SIL code, software weakness analysis can be performed well through the SIL.

In the final step, we generated the CFG of the SIL code and the SIL code by performing the conversion to the SIL through basic type inference using CFG information and assembly code. Figure 7 shows that the contents of block 2656 in Figure 6 are converted to SIL code. You can see that the assembly code has been converted to SIL code with equivalent semantics, and the reasoning behind the basic data type for the used variable.

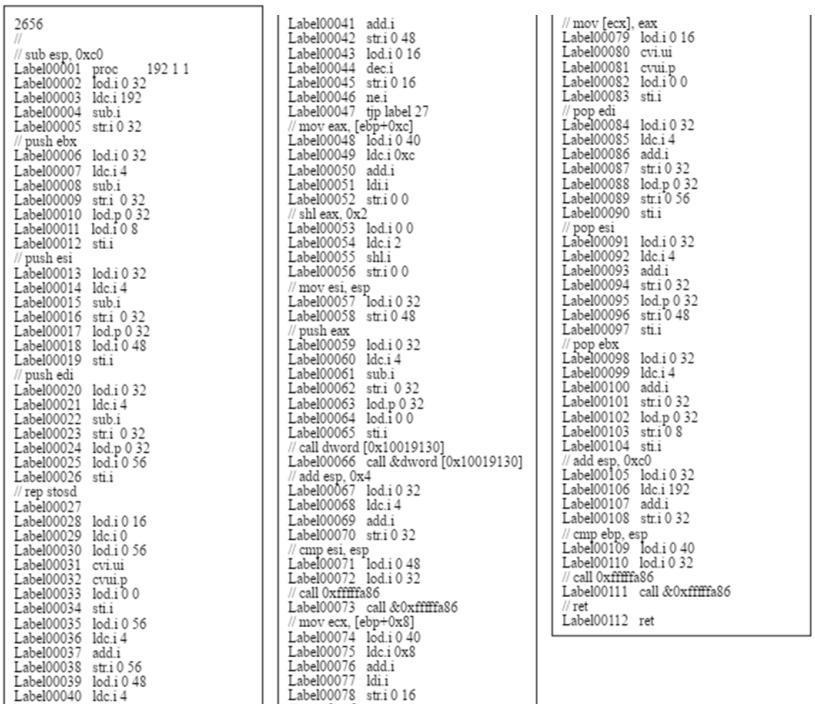

Figure 6. SIL Code of wrapperMalloc Function

Finally, the CFG of the SIL code is also represented by the flow as shown in the left side of Figure 7, and the connection of the block is the same as the CFG of Figure 5. As described above, since one assembler code is converted into a plurality of SIL codes, the entire code length becomes very long, so it is difficult to express the contents of the entire code, so blocks 2822, 2857, 2597 and 2912 have been enlarged. The right side of Figure 7 shows that each code is converted.

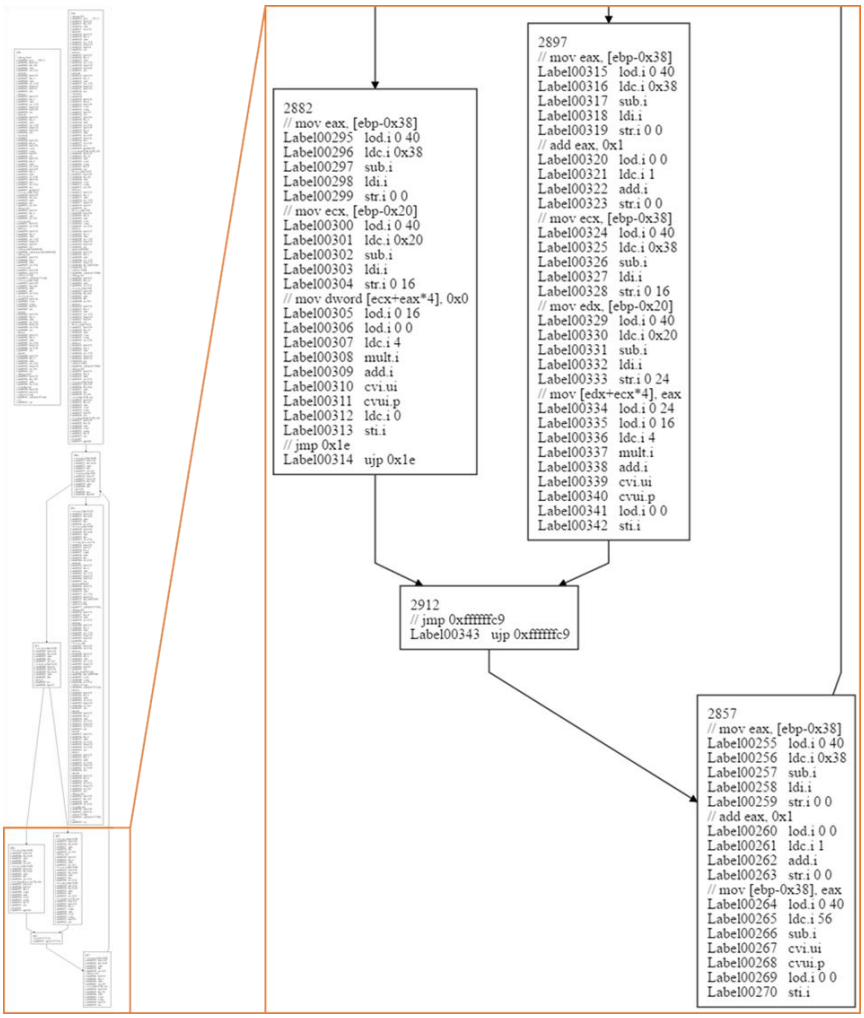

Figure 7. A CFG Example with SIL Code

5. Conclusions and Further Researches

We have selected and modified the SIL for techniques that utilize intermediate languages, which is one of the techniques for statically analyzing software weakness inherent in binary code. We also convert the binary code into a modified SIL for use in software weakness analysis and generate a CFG to prove that the semantics of the transformed intermediate language are valid. And analyzed the results using the visualization tool we created. As a result, we can confirm that the actual meaning of the binary code is maintained and converted into the equivalent SIL.

However, when converting from binary code to intermediate language, the inference about the data type is made only for the basic data type, so that the inference about the complex data type is not done and the whole code is converted as the CISC code x86 / 64 assembly language into RISC type SIL code. The problem is that the overall length of the code is too long.

Future research will focus on more accurate data type inferences for basic data types and complex data types from binary codes and extracting only the parts needed for weakness analysis to reduce the length of the SIL.

Acknowledgments

The authors gratefully acknowledge the financial support provided by Defense Acquisition Program Administration and Agency for Defense Development under the contract UD160035ED.

References

[1] N. Mehta, "The Heartbleed Bug", **(2014)** April.
[2] S. Chazelas, "The Shellshock vulnerability," **(2014)** September.
[3] B. Möller, T. Duong and K. Kotowicz, "This POODLE bites: exploiting the SSL 3.0 fallback," **(2014)**.
[4] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, ... and E. Käsper, "DROWN: Breaking TLS using SSLv2," Proceedings of the 25th USENIX Security Symposium, AUSTIN, USA, **(2016)** August 10-12.
[5] GRAMMATECH, "Find Defects in Third-Party Code," http://www.grammatech.com/products/binary-analysis
[6] GRAMMATECH, "Eliminating Vulnerabilities in Third-party Code with Binary Analysis," http://www.grammatech.com/products/codesonar
[7] A. Mycroft, "Type-based decompilation (or program reconstruction via type reconstruction)," Proceedings of the 8th European Symposium on Programming Languages and Systems, Amsterdam, Netherlands, (1999) March 22-28.
[8] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, and P. Narasimhan, "Recovering c++ objects from binaries using inter-procedural data-flow analysis," Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop, San Diego, USA, **(2014)** January 22-24.
[9] K. Yoo and R. Barua, "Recovery of object oriented features from C++ binaries," Proceedings of the 21th Asia-Pacific Software Engineering Conference (APSEC), Washington, USA, **(2014)** December 01-04.
[10] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," Proceedings of the 4th International Conference on Information Systems Security, Hyderabad, India, **(2008)** December 16-20.
[11] G. C. Necula, S. McPeak, S. P. Rahul and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," Proceedings of the 11th international Conference on Compiler Construction, Grenoble, France, **(2002)** April 8-12.
[12] T. Dullien, and S. Porst, "REIL: A platform-independent intermediate representation of disassembled code for static code analysis," Proceeding of CanSecWest **(2009)**.
[13] S. Cesare, and X. Yang, "Wire-A Formal Intermediate Language for Binary Analysis," Proceedings of IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Liverpool, UK, **(2012)** June 25-27.
[14] J. Viega, J.T. Bloch, Y. Kohno and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code," Proceedings of IEEE 16th Annual Conference on Computer Security Applications, New Orleans, USA, **(2000)** December 11-15.
[15] S. Rawat and L. Mounier, "Finding buffer overflow inducing loops in binary executables," Proceedings of IEEE 6th International Conference on Software Security and Reliability (SERE), Gaithersburg MD, USA, **(2012)** June 20-22.
[16] J.M. Lee, H.W. Kim and W.H. Ahn. "BinaryReviser: A Study of Detecting Buffer Overflow Vulnerabilities suing Binary Code Patching," Proceedings of the Korean Institute of Information Scientist and Engineers Conference, Gyeongju, South Korea, **(2011)** June 29-July 1.
[17] T. Wang, T. Wei, Z. Lin and W. Zou, "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution," Proceedings of 16th Network and IT Security Symposium, **(2009)** San Diego, USA, February 8-11.

