**SCORE: Source Code Optimization & Reconstruction**

JAE HYUK SUK, YOUNG BI LEE, AND DONG HOON LEE, (Member, IEEE)

Graduate School of Information Security, Korea University, Seoul 02841, South Korea

Corresponding author: Dong Hoon Lee (donghlee@korea.ac.kr)

This work was supported in part by the Military Crypto Research Center funded by Defense Acquisition Program Administration (DAPA) and Agency for Defense Development (ADD) under Grant UD170109ED.

**ABSTRACT** The main goal of obfuscation is to make software difficult to analyze. Although obfuscation is one useful method to protect programs, the ability to analyze malware is greatly reduced if used for malicious purposes. The obfuscation technique is most applicable at the binary level, but it can also be applied at the source code level. Although source-level techniques can be applied regardless of the target platform, these are often optimized and eliminated during compilation. However, when control-flow obfuscation is applied at the source code level, removal is not possible. When applied for malicious purposes, the ability to analyze the source code and compiled binary code is greatly reduced. To date, no research has presented a method that increases the readability of source code or the ability to analyze compiled binaries via optimization at the source level. In this paper, we select a very powerful obfuscation tool that provides options, including control-flow obfuscation, at the source level. The result of our research is a tool that outputs optimized source code and performs control-flow reconstruction as preprocessing, which increases readability even when control-flow obfuscation has been applied. The results also suggest an improvement in the ability to analyze binary code. As a result, more than 70% of the source code can be optimized at the source level, and the control-flow graph can be serialized. The optimized source code compiles more concise binary code even if no compiler optimizations are applied. Finally, the paper concludes by presenting the results of a module that prevents deobfuscation through code tampering (preventive obfuscation) at the source code level.

**INDEX TERMS** Control-flow obfuscation, control-flow reconstruction, source code obfuscation, source code optimization, source code anti-tampering.

**I. INTRODUCTION**

As software becomes increasingly important in modern society, infringements of software intellectual property rights (IPR) and attacks on software vulnerability are becoming grave concerns. A malicious analyst (attacker) infringes on the IPR of software through static analysis or dynamic analysis and bypasses license restrictions by modifying it. To solve this problem, recent research focuses on preventing reverse engineering, and code obfuscation is one of the technical means through which this can be achieved [1]. Code obfuscation is a code-transforming technique that makes the code challenging for an attacker to understand while maintaining functionality of the original code.

On the other side of this issue is malware. When malware authors apply obfuscation techniques to malware, it not only reduces the detection rate of antivirus software but also makes malware analysis extremely cumbersome [2]. Therefore, it is equally necessary to examine deobfuscation and optimization of obfuscated programs. In particular, existing commercial obfuscation tools facilitate easy application of various obfuscation techniques, but there are no commercial deobfuscation tools. Although some non-sale scripts (e.g., Themida/WinLicense Ultra Unpacker, VMProtect Ultra Unpacker, PolyPack Unpacker etc., [3], [4]) to deobfuscate the obfuscated program are available, these can be useless under some situations such as obfuscator version upgrade or protection option enhance. In particular, since there are many obfuscation tools that significantly enhance the obfuscation (In case of Themida, method of analysis is entirely changed as of version up) according to the version upgrade, generality must be emphasized in order to present a commercial deobfuscation tool, which means that it takes a lot of time for one program deobfuscation and it is difficult to use in the field.

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

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/
In conclusion, this means that any analysis of obfuscated malware can be seriously delayed.

This paper considers two locations for obfuscation techniques in software: at the source code level and the binary level—or both [5]. When distributors need to disseminate source code of a piece of software, it is likely that source code, or source-level, obfuscation will be applied. However, if it is not indicated that the source code needs to be distributed, the distributors normally apply binary-level obfuscation. Source code obfuscation techniques [7] can transform the source code itself, irrespective of the target architecture. Some information is available in only source code level, unlike binary code level, and this information facilitates obfuscation techniques. Although there are techniques that can indeed be applied at the lower (binary) level, these are still limited, in that they can only be applied at the source code level (i.e., instruction overlapping). Another limitation of source code techniques is that obfuscated parts are partially removed in the process of compiling the source code.

In contrast, binary-level obfuscation techniques are specific to the target architecture. As some information is lost in the process of compiling binary code, binary-level obfuscation, through binary rewriting, poses challenges. Despite this difficulty, binary-level obfuscation techniques can be designed to require precise target addresses or assembly code, unlike source-level techniques. Moreover, binary is the last phase of software development. This suggests a clear advantage of applying obfuscation at the binary level, in that obfuscated parts would not removed by an subsequent process, such as code optimization.

Source-level obfuscation techniques differ slightly from binary-level obfuscation techniques. First, binary obfuscation can be applied to protect the IPR of software, and while obfuscation of source code is similar in purpose, it requires the vendor to supply the source code. In this case, the vendor must obfuscate the source code because it is necessary to hide the confidential information it contains. Similarly, because JavaScript code is exposed on web browsers, the engineer can use source-level obfuscation to protect it. This is also true for source-level optimization. If source-level obfuscation techniques are applied to malware, it is difficult to analyze functionality, suggesting that analysts must study and apply source-level optimization. If a strong obfuscation technique is applied to the source code but is not optimized well by the compiler, obfuscation is not removed during the compilation process, and the difficulty of analysis is propagated and maintained. For analysts, it is necessary to remove obfuscation techniques that are not optimized well during compilation, while increasing the readability of the source code.

In this paper, we propose our automatic source-level optimization tool, named SCORE: Source Code Optimization & REconstruction. SCORE can quickly cope with source-level obfuscation applied in a malicious scenario.

A. CONTRIBUTIONS

For this paper, we selected an obfuscation tool that provides obfuscation options at the source code level, and we analyzed the obfuscated program at the source code level. The selected tool provides control-flow obfuscation techniques, which are rarely provided at the source code level. Through our experiments, we proved that this tool optimizes source code and performs control-flow reconstruction, which improves the readability of source code when control-flow obfuscation has been applied. In addition, out results prove that readability is improved even for binary code generated during compilation. This paper also demonstrates that our proposed tool guarantees a higher degree of protection than that of existing source-level tools that provide only layout obfuscation or data obfuscation.

This paper makes the following contributions:

- We choose a source-level obfuscation tool that can be applied to C/C++ and analyze its features and options. We also present the implementation results of the tool SCORE that can apply optimization to the C/C++ source code. The output of SCORE is more intelligible than source code optimized by a compiler. This is because the compiler’s optimization module is a middle-end component that is applied at the intermediate representation (IR) level, whereas SCORE is an independent tool that can be applied at the source code level. When optimization is applied to the IR, it is not trivial to automatically translate the optimized IR back to the recompilable source code. Therefore, although IR with optimization is more efficient than source code, it is not necessarily easier to understand.

- SCORE, the optimization and reconstruction tool proposed in this paper, provides high readability even when control-flow obfuscation has been applied to source code because it performs control-flow reconstruction statically & dynamically. Since the compiler optimization module cannot perform this reconstruction process, it is still difficult to understand or analyze the control-flow of the compiled binary code even when optimization is completed at the IR level. Preprocessing of the control-flow reconstruction results in more patterns that can be optimized or removed. We ultimately show that SCORE can optimize source code and binary code more aggressively than fully optimized binary from the compiler optimization module, even when SCORE output source code is not optimized by the compiler optimization module.

- In order to restore original source code from obfuscated source code, SCORE must be able to tamper the source code during analysis, reconstruction, and optimization. As such, a self-checkable source code must be generated and inserted to strengthen source code level obfuscation techniques. In this paper, we present the results of
generating and applying code blocks that verify whether the source code is tampered for analysis purposes.

B. PAPER ORGANIZATION

The remainder of this paper is organized as follows. In Section II, we present related works on source-level obfuscation and compiler structure. Section III first outlines and analyzes existing source-level obfuscation tools that provide layout and data obfuscation options, then we analyze one tool that provides control-flow obfuscation options. Section IV describes the features and options available in the source-level obfuscation tool and describes control-flow reconstruction and optimization. Section V discusses the challenges of analyzing the source-level obfuscation tool and related countermeasures. Section VI presents our implementation results, and Section VII evaluates the performance of our proposed tool, SCORE, as compared with other compiler optimization modules. Finally, Section VIII presents countermeasures, and Section IX presents the conclusion of our paper.

II. RELATED WORKS

In this section, we describe source-level obfuscation techniques. We also describe compiler structure as a technique that can be compared with obfuscation.

A. SOURCE-LEVEL OBFUSCATION TECHNIQUES

The compiler optimizes code by compiling obfuscated source code to the binary. That is, the source code is high-level code, and parts may be removed or optimized, thus losing much information in the process of generating low-level code from the input.

Obfuscation techniques are divided into four categories [1]: layout obfuscation (transformation or elimination of additional elements: variable name, comment, etc.), data obfuscation (transformation of data structures), control-flow obfuscation (transformation of control-flow: loops, branch statements, etc.), and preventive obfuscation (analysis prevention routines: anti-debugging, anti-tampering, etc.). The compilation of source code with layout obfuscation loses all potency at the binary level. The reason for this is that, regardless of whether or not layout obfuscation is applied, the source code always compiles to the same binary code without variable names or comments. In the case of data obfuscation, simple data obfuscation (e.g., value modification) is optimized by the compilation process and potency is lost. However, complex data obfuscation (e.g., array reconstruction) is not optimized well. Finally, in control-flow obfuscation, potency is not lost during compilation because it is not optimized well by general compiler optimization techniques.

To date, only a few studies describe the need for obfuscation at the source code level. First, [8] suggests a controlled network environment and assumes that protected software (source code) is operating in an untrustworthy environment. Reference [8] presented the results of implementing a compiler pass that could apply control-flow obfuscation to target source code. The study showed performance overhead and protection strength of the obfuscated program via this pass, and became the first research paper to suggest the necessity of studying source-level obfuscation techniques. Although it is not (original) source code to (obfuscated) source code transformation research, this landmark study clearly indicated the need for further research in this area.

Subsequently, [9] presented the effect of source-level obfuscation on binary code. As described above, obfuscation is at least partially removed during compilation optimization. Reference [9] implemented a framework for source code obfuscation and suggested that the technique becomes a formidable obstacle in analyzing binary if strong obfuscation is applied only to the source code. In other words, applying a strong source-level obfuscation tool means that binaries with high protection strength can be generated even after compilation.

Tigress [10] and ASPIRE framework [5] are source code level obfuscation tools published in academic research. These are source-to-source obfuscators that also contain control-flow obfuscation techniques. Both Tigress and ASPIRE framework can obfuscate C or C++ language, and in particular, ASPIRE can insert a code guard in the protected target source code. Code guard of the obfuscated source code can resist injudicious source code tampering or modifications. However, the code guard is exposed as a Marker form and it is needed to compile for activation. As such, although Tigress and ASPIRE framework can protect the source code with strong control-flow obfuscation techniques, the attacker can still analyze the source code through source code tampering even if only the source code is exposed to the attacker.

B. SOFTWARE OBFUSCATION TECHNIQUES

Except for research limited to the source code level, software obfuscation and deobfuscation methods are continuously being studied. Especially, there is much scholarship regarding generic techniques to cope with obfuscated software containing multiple options. As one example, [25] analyzed obfuscated software using symbolic execution. Results are presented for various obfuscation tools, such as Themida and VMProtect, among others. However, [25] did not analyze the behavior of obfuscation tools. Therefore, the only tracing time for analysis is a maximum of 16 minutes with Themida. Although [25] successfully illustrates generic and effective results, there was a clear need for additional study of optimization performance overhead.

Another major research topic on software analysis is Dynamic Symbolic Execution (DSE). In this area, researchers explore automatic software analysis tools based on the DSE technique. Reference [26] suggests such a technique that facilitates the analysis of strongly obfuscated software. It can also perform a near restoration of the control-flow graph to the original program. Unfortunately, this specific technique cannot be used in the field because it takes a significant
amount of time to analyze the obfuscated software. Thus, additional optimization research is required in this area as well.

Next, although it is not a deobfuscation technique but an obfuscation technique, the Mixed Boolean-Arithmetic (MBA) technique [24] has also been considered as a powerful data obfuscation technique. In particular, it is known to make optimization with DSE difficult thanks to its strong resilience. Still, only limited research on MBA expression simplification is available to date [27], [28]. Simplifying requires a great amount of time, and the possibility of simplification varies depending on the complexity of the original MBA expression.

As described above, the recent trend in obfuscation analysis technique research is based on broad generality and minimal assumptions. However, these studies share a common limitation, which is that the time required for deobfuscation is very large (averaging more than 30 minutes). Although deobfuscation is a one-time process, it is an opportune point to respond to ever-increasing malware.

Therefore, even if generality is somewhat downgraded or assumptions are needed (applied to the specific group of obfuscation tools), it is also necessary to study techniques that can be directly applied in the field. Recently, studies such as [29] have attempted to meet such needs. Reference [29] can deobfuscate a number of binary protectors, and its deobfuscation time is greatly reduced (to about 1 minute). Owing to this, we also suggest source code level optimization that can be applied in the analysis field in this paper.

Although it is not a source-to-source deobfuscation technique, the technique for optimizing it by converting the binary to LLVM-IR has also been proposed [30]. As it can be applied to various types of target architectures, so it has a contribution that it can provide high versatility. However, it is noted that optimization is not supported for some powerful control flow obfuscation techniques such as Virtualization of Tigress.

Recently, researches are often studied as an anti-DSE technique to prevent the DSE technique. Reference [31] proposed a systematic review of anti-DSE techniques. The results of researches conducted up to now have been classified according to the characteristics, and by describing the limitations of previous researches, the direction of future research was presented. In particular, it was pointed out that the experiment for verification of the proposed anti-DSE was insufficient, and also many implementations are not available.

In [32], the resilience of obfuscated C code was measured using program slicing. By measuring the resilience, they measured the degree of resistance to automated deobfuscation tools. The experimental method is to insert a marker indicating the scope of program slicing in the source code and then obfuscate it using Tigress. After that, deobfuscation was performed by applying program slicing, and the results were presented.

At first glance, the above research seems almost the same as our source-to-source deobfuscation, in that obfuscation and deobfuscation are done at the source code level. However, the above research has a difference in that it is possible to perform deobfuscation only by inserting a marker in the original source code. In other words, it means that attackers who can acquire only the obfuscated source code are difficult to utilize the above research. Also, it was mentioned in the above research that the strong obfuscation (e.g., control-flow obfuscation) has very high resilience in the source code level, so the source code level deobfuscation technique must be studied separately.

C. COMPILER STRUCTURE AND PRINCIPLES

A compiler is a program that translates high-level programming language into low-level machine language that a computer can interpret or execute [11]. The code written in programming language is called source code, and the machine language generated by the compiler is called binary code (or an executable file). The compiler generally consists of three stages: front end, middle end, and back end. At the front end, the source code is input, and at the back end, the binary is output. The structure of a traditional compiler is shown in Figure 1.

The front end parses the source code as input and generates an IR corresponding to the source code. The middle end generally performs optimization and generates an optimized IR to improve the performance of the generated binary. Finally, the back end generates executable files for the target architecture from the optimized IR.

When using IR, optimization techniques can be applied independently of the target architecture, and it facilitates the design of portable compilers, making its use advantageous. However, the readability and comprehensibility of the source code are reduced after converting to IR.

III. SOURCE-LEVEL OBFUSCATION TOOLS

In this section, we describe obfuscation tools that can be applied at the source code level\(^2\) and compare the control-flow graph of obfuscated source code with that of the original source code. The experiments in this section show

\(^2\) As these are the subject of our experiments, obfuscation tools are only briefly outlined in this paper for comparison purposes, but are not analyzed in detail.
how much of the source-level obfuscation is removed during compilation.

We selected SD [12] and SX [13] as target obfuscation tools for our experiments. Each provides options for layout obfuscation and data obfuscation, and each tool applies layout obfuscation, including comment removal, variable name modification, function name modification, and blank and newline character removal to the source code. At the same time, data obfuscation (e.g., string encoding and variable value modification) is applied. Both tools apply only simple data obfuscation that modifies variable values. For example, neither of the tools modifies the structure of an array in the source code.

Figure 2 shows the result of applying SD and SX to source code that performs binary-coded decimal (BCD) conversion. The obfuscation techniques mentioned above are applied. In particular, transforming variables and function names into similar forms (combinations of 0, O, 1, and I) and removing comments is very simple, but effectively reduces the readability of the source code.

However, because data structures such as arrays are not transformed, and the control-flow is not transformed at all, the binary files generated by the compilation of all three variants are the same. The control-flow graph in each case is shown in Figure 3, and the results of applying the hash function to the executable code section are identical. In other words, when the commercial source code obfuscation tool is applied to the target source code, the potency of the source code increases greatly, but both the obfuscated source code and original source code generate the same binary during compilation [14]. Also, at the source code level, it is not burdensome to increase the readability of variable names, code layouts, and split variable values. Of course, deobfuscation cannot restore deleted comments or semantics of variable names.

In other related works, authors have mentioned the necessity of control-flow obfuscation techniques to protect not only the source code but also the optimized binary. However, to date, very few commercial source code obfuscation tools can apply control-flow obfuscation at the source code level. SF [15], one tool that provides control-flow obfuscation options, hinders easy analysis via control-flow flattening [16] and virtualization [17]. These two obfuscation techniques modify the control-flow graph aggressively to make it challenging to understand the order of execution of the program. In particular, these techniques are highly resilient, as they are composed of an interpreter-like structure [18]. This means that obfuscation applied at the source code level is nearly untouched during compilation, and thus, analysis of the compiled binaries becomes quite challenging.
TABLE 1. List of terms.

| Terms             | Description                                                                                                                                 |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| Code Block        | Source code unit generated by SF, composed of tens to hundreds of lines. The dispatcher can also be referred to as one of the code blocks.     |
| Label             | This is the component on the top line of the code block. It appears in the form of an L#: (Ex: L131).                                           |
| Dispatcher        | This is the component that selects the code block to be executed next. It appears in the form of a switch-case statement.                     |
| Dispatcher Variable | This is the component used in the dispatcher. According to this value, control is moved to the corresponding code block. The value of the dispatcher variable is modified in the code block. |

IV. SOURCE CODE OPTIMIZATION AND CONTROL-FLOW RECONSTRUCTION

In this section, we present our analysis of SF features, obfuscation options, and control-flow obfuscation techniques. Based on this analysis, we performed a control-flow reconstruction and designed appropriate optimization tools at the source code level. The terms used throughout this paper are listed in Table 1.

A. TARGET SOURCE CODE OBFUSCATION TOOL

SF is a commercial source code obfuscation tool that provides control-flow obfuscation techniques. This tool differs from the two tools described above, in that it provides obfuscation on a function unit. (Although SF also provides layout obfuscation and data obfuscation, this paper does not describe those options in detail because we are focused on control-flow obfuscation.)

SF provides control-flow flattening and virtualization, which are shown in Figure 4.

As virtualization and control-flow flattening techniques are complex in structure, and optimization is rarely performed, the control-flow graph is cumbersome to analyze even when the compiler optimization is performed. Figure 5 shows the output of the control-flow graph after the “HelloWorld” source code was obfuscated by SF and compiled. The highest level of optimization (−O3, Optimization Level 3) was applied as a compiler option.

In this paper, we propose utilizing the control-flow reconstruction process as a preprocessing process, as shown in Figure 6. In SF control-flow obfuscation techniques, the dispatcher is composed of a switch-case statement. The entire control-flow can be serialized by performing code block realignment (in order of execution), control transfer statement (i.e., a goto statement) removal, label removal, and finally dispatcher removal. This process becomes the basic algorithm of control-flow reconstruction that we present in this paper.

B. ADDITIONAL OPTIONS FOR CONTROL-FLOW OBFUSCATION

As described in the previous subsection, we must perform dispatcher removal after serializing the control-flow to optimize the SF control-flow obfuscation. However, as there are additional obfuscation options in SF that can delay the process of analyzing the control flow and obfuscation structure itself, the process of analyzing each option must precede. In this subsection, we describe SF options associated with control-flow obfuscation. SF provides additional options that can be roughly categorized as label obfuscation, code block obfuscation, and internal obfuscation.

- Label Obfuscation

A label is a component that appears on the first line of a code block. It is used for dividing the code into blocks and as an index of the code block to be selected by the dispatcher. When the label is obfuscated by SF options, new conditions (cases) are added to the transfer control for each label. However, because the added conditions

FIGURE 3. Control-flow graph of compiled binary (SD and SX).
cannot be satisfied, control transfer to the code block is impossible. That is, the newly added conditions are fictive. Therefore, label obfuscation corresponds to the unreachable code insertion; during analysis, it is taxing to grasp the correct control-flow without removing this unreachable code. An example of label obfuscation is shown in Figure 7. Var value in the example is never calculated as 604, 796, 504, or 239.

- Code Block Obfuscation

A code block is responsible for the functionality of obfuscated source code in a flattened structure. When the code block is obfuscated by SF, fictive conditions and labels are added along with fictive code blocks that correspond to the added labels, so it appears as if the control can transfer to the added fictive code blocks. In practice, however, these do not execute at all, for the same reasons as label obfuscation. Therefore, this also becomes an issue of unreachable code insertion; it is difficult to analyze this code because of the explosive increase in code volume. An example of code block obfuscation is shown in Figure 8 (the left side is the dispatcher and the right side is the code block). Var value in the example is never calculated as 604 and 796, so Label2 and Label3 are never executed. In addition, although this example simply copies code blocks of functionality A, techniques such as code block modification, merging, and splitting can also be applied at the same time.

This is similar to label obfuscation, in that additional fictive conditions are added, but much more unreachable code is generated because fictive labels and code blocks corresponding to fictive conditions are generated together.

- Internal Obfuscation

Select options that obfuscate statements inside a code block make it impossible for analysts to perform
efficient data flow analysis. Such options typically include dead code insertion and variable value splitting. With these techniques, the functionality of each code block is not destroyed, but the size of the code block greatly increases. Unlike code block obfuscation, the added code is executed, but it is meaningless (dead code). In this case, analysts should apply optimization instead of removing the code indiscriminately. An example of internal obfuscation is shown in Figure 9. The third line (reallocating the variable value) can be eliminated by dead code elimination, and the fourth line
FIGURE 9.  Internal obfuscation example.

(splitting the variable value) can be precomputed (optimized) by constant propagation and constant folding.

V. CHALLENGES

SF is the source code level obfuscation tool, which simultaneously performs obfuscation transformation in a function unit. In addition, the somewhat special structure of this tool when applying control-flow flattening (i.e., dispatcher & code block) renders it virtually impossible to reconstruct the control-flow.

In this section, we describe the challenges of analyzing and deobfuscating programs that have been obfuscated by SF.

A. DISPATCHER VARIABLE

Internal obfuscation can be removed by general compiler optimization techniques, but code block obfuscation and label obfuscation cannot be easily removed by the compiler. This is because it is difficult to know the execution order of code blocks at compile time. In particular, this also applies to ascertaining the value of the dispatcher variable (used for code block selection) at compile time, which is read from outside the protected function. Therefore, it is necessary to optimize the entire source code from a large unit to a small unit in order to reconstruct the control-flow, eliminate unreachable code blocks and dead code, and apply internal code optimization.

Static analysis is not enough to reconstruct the control-flow via the actual execution order of code blocks. This is because control-flow flattening prevents any meaningful analysis here, and dynamic analysis becomes necessary to acquire the execution order of code blocks. To this end, we designed a label extraction module to obtain a label list of executed orders as shown in Figure 10. After this point, the control-flow reconstruction module places code blocks in order of execution.

B. BRANCH STATEMENT RECONSTRUCTION

SF can also obfuscate branch statements, and if only a dynamic analysis for reconstruction were applied, then the functionality of obfuscated program would be destroyed because our tool SCORE is based on execution results. As a simple illustration of this, the source code on the right side of Figure 11 can be obfuscated to the left side of Figure 11.

In dynamic programs, execution flow is determined at run time as shown in Figure 11. To optimize such source code, a module that can also handle dynamic programs is needed. Specifically, the branch statement reconstruction module is needed before the control-flow reconstruction module as shown in Figure 12.

If the branch statement reconstruction process is not preceded, the execution path is fixed during the control-flow reconstruction process. This means that even if the user input value is entered to the optimized source code later, the correct result cannot be acquired.

In the simplified example of Figure 12, the next code block is selected at the end of the executed code block, not by the dispatcher. However, the branch statement can be reconstructed by a similar method even in structures where the dispatcher exists. By reconstructing the branch statement, it is possible to reduce the number of labels to be extracted and structurally improve readability.

C. FUNCTION EXECUTION PATTERN

Because SF obfuscates source code in function unit and SCORE reconstructs code blocks dynamically, the code block log that is traced by the label extraction module can include useless code blocks depending on the execution pattern of the protected target function (i.e., multiple executions). Also, the target function can be executed on a different execution path if the function has a branch statement. In addition, there may be common code blocks executed (i.e., common code block) in the target function that are executed multiple times, and the control-flow reconstruction module should reflect this. Otherwise, redundant code blocks will be duplicated, reducing the efficiency of optimization. As shown in Figure 13, inserting control-flow controllers allows a target function that is executed multiple times to be reconstructed. The loop identification algorithm [19] is applied to design the module, and extraction and merging of common code blocks is performed.

VI. IMPLEMENTATION

In this section, we outline the SCORE structure and its algorithm, as designed and implemented in this paper. The structure of SCORE is illustrated in Figure 14. SCORE is implemented in C language with about 2000 LOC. The obfuscated source code parsing module and optimized source code

3In Figure 12, it can be calculated heuristically that L34 is a trivial label that is replaceable with a break statement during the reconstruction of the branch statement.
generation module are not described separately because their functionalities are highly intuitive.

A. REGULAR EXPRESSION MODULE
This is a module used to extract specific obfuscation patterns or code patterns from obfuscated source code. Because obfuscation patterns are not standardized, all variants of obfuscation patterns would need to be stored in the database if this module did not exist. Owing to this, we used a regular expression to specify the basic form of an obfuscation pattern and its variants. In addition, because obfuscation is not applied to the entire source code but to the user-selected target function unit, this regular expression can be used to efficiently extract the function for analysis.

B. LABEL EXTRACTION MODULE
This module extracts the execution order of code blocks in the obfuscated source code. First, this module inserts the label print application programming interface (API) below each label. This serves to output the label of code blocks actually executed at runtime. Code blocks corresponding to the labels are rearranged by order of execution to perform control-flow reconstruction as described above.

C. BRANCH STATEMENT RECONSTRUCTION MODULE
This module reconstructs branch statements if they exist in the obfuscated target function with an obfuscated form as
described in the above section. Most branch statements can be reconstructed by code block parsing and merged with a `goto` label statement. However, some cases of branch statements, such as the one shown in Figure 12, should be heuristically implemented (e.g., `if` statement + `goto` statement (to `if` code block) in `if` code block ⇒ `for` statement).

D. CONTROL-FLOW RECONSTRUCTION AND CODE OPTIMIZATION MODULE

This module rearranges code blocks in the order of execution using labels extracted from the preceding module and applies elimination and optimization to the serialized code. It consists of three submodules: one each for control-flow reconstruction, source code elimination, and source code optimization. The control-flow reconstruction submodule is executed only once during the first round, while the source code elimination and optimization submodules are repeatedly executed until neither removable patterns nor optimizable patterns can be found.

1) CONTROL-FLOW RECONSTRUCTION MODULE

One feature of source code with applied control-flow obfuscation is that all code blocks start with a label. Therefore, code blocks are rearranged in order of label execution. As all code blocks jump to the dispatcher after their execution ends, the most executed code block, which has about half the total number of code block executions, corresponds to the dispatcher itself. This is the code block that should be removed in this module or regenerated as control-flow controllers. In addition, as the code blocks are rearranged in order of label execution, the unreachable code blocks are automatically eliminated [20].

As the dispatcher is removed in this process, all code blocks are inserted with their respective labels and control transfer statements (i.e., `goto` dispatcher statement) removed. However, there are cases where the dispatcher transfers control to a specific code block two or more times, so the amount of source code after completion of this module cannot always be reduced—in fact, it might even be increased.

2) SOURCE CODE ELIMINATION MODULE

A source code elimination module identifies removable patterns and removes them at the source code level. This module applies dead code elimination, meaning that it removes code that actually executes but does not affect program
functionality. For this paper, we focused on identifying and removing the variable value reassignment code.

Because dead code elimination is well known as a compiler optimization technique [20], a description of the detailed algorithm is omitted in this paper.

3) SOURCE CODE OPTIMIZATION MODULE

A source code optimization module identifies optimizable patterns and optimizes them at the source code level. In this module, constant propagation, copy propagation, array value propagation, and constant folding can be applied to the source code. For the purposes of our research, we focused on identifying and optimizing variable value-splitting patterns.

Because the constant propagation technique, the copy propagation technique, array value propagation technique, and constant folding technique are widely seen in compiler optimization processes [20], detailed descriptions of the algorithms are omitted from this paper.

E. SCORE ALGORITHM

The SCORE algorithm is implemented according to the above overview, and detailed module functionality is shown as Alg 1. In the SCORE tool, the label extraction module compiles and executes the source code dynamically with the label print API inserted. All other modules correspond to static modules that parse and process the source code. The SCORE execution process and logical relationships are presented in the flowchart shown in Figure 16. In the flowchart, the paranthetical portions indicate module names of SCORE.

An important point in the process of reconstructing source code is to understand that the code blocks are reassigned in order of execution. To do this, it is necessary to analyze the label and remove the code block corresponding to the dispatcher. The control-flow flattening structure of SF does not perform functionality immediately according to condition (i.e., dispatcher variable) in the dispatcher, but moves control to the code block corresponding to the condition, after which it executes the functionality. Due to these structural characteristics, the pattern of Code Block # → Dispatcher → Code Block # → Dispatcher → … is generated in the label log by the label extraction module. Therefore, the label executed the greatest number of times represents the label of the code block corresponding to the dispatcher, and that code block is automatically excluded from insertion during the reassignment process.

In addition, the algorithm presented in this paper can be considered somewhat applicable to Tigress, which provides control-flow flattening at the source code level. Source code that is obfuscated by Tigress is shown in Figure 15. The actual structures of control-flow flattening in SF versus Tigress differ slightly. There are neither jumps to code blocks using labels, nor Goto statements that return to the dispatcher after executing code blocks as shown in Figure 15. However, it is noteworthy to mention that break statements can return control to the dispatcher.

In other words, it can be considered that the functionality and dispatcher are combined. In this case, the proposed control-flow reconstruction algorithm can be applied as is. The difference is that the label extraction function is not inserted into the code block, but rather into the Case.

VII. EVALUATION

In this section, we evaluate the performance of the SCORE implementation. The results show the degree of optimization performed by SCORE in terms of the number of source code lines in the optimized code and the elapsed time for optimization. For SCORE evaluation, we select C/C++ compilers and compare the optimization done by SCORE with the optimization techniques provided by selected compilers; we present the optimized function size, number of optimized function assembly lines, and the control-flow graph. All experiments were conducted on Windows 10 on an Intel i5 3.30 GHz processor with 16 GB of RAM, with obfuscated source code produced by SF.

A. SCORE RESULT

In Table 2, we present the SCORE optimization results for source code implemented in C/C++. The elapsed time refers to the time required until the final round of the optimization process but excluding the time required to output the source code generated as the intermediate result for each round. The number of source code lines in the result indicates the number of function lines to be optimized (i.e., target function), not the entire source code. Also, the Optimization Ratio of result table can be calculated as follows.

\[
\text{Optimization Ratio} = \frac{1}{1 - \frac{(After \div Before)}{100}}
\]

For example, Optimization Ratio of CRC32 can be calculated by \(1 - (433 / 1760) \rightarrow 75.39\%\).
Algorithm 1 SCORE Algorithm

Input: ObfuscatedSourceCode
Output: OptimizedSourceCode

procedure (SCORE)
    Round = 0
    while Round == 0 or SourceCodeLine [Round-1] != SourceCodeLine [Round] do
        ObfuscatedCode = FileOpen(ObfuscatedSourceCode, Read)
        OptimizedCode = FileOpen(OptimizedSourceCode, Write)
        FunctionCode = TargetFuncIdentification(ObfuscatedCode)  \> Obfuscated Source Code Parsing
        TotalLine = LineOf(FunctionCode)
        if Round == 0 then  \> Control-Flow Reconstruction Performed Only Once During First Round
            LabelIndex = 0, BranchIndex = 0
            LabelLogCode = FileOpen(LabelPrintedSourceCode, Write)
            for Line = 0 → TotalLine do
                Statement[Line] = StatementParser(FunctionCode)
                if isLabel(Statement[Line]) then
                    LabelList[LabelIndex] = Statement[Line]
                    FileWrite(LabelLogCode, LabelList[LabelIndex] & LabelPrintAPI)  \> Label Extraction
                end if
                if isBranch(Statement[Line]) then
                    ReconstructedBranch[Line] = BranchReconstruction(Statement[Line])
                    FileWrite(LabelLogCode, ReconstructedBranch[Line])
                end if
                if !isBranch(Statement[Line]) && !isLabel(Statement[Line]) then
                    FileWrite(LabelLogCode, Statement[Line])
                end if
            end for
            LabelLog = Compile&Execution(LabelLogCode)
            Dispatcher = MaxBlock(LabelLog)
            CodeBlockReassignment(OptimizedCode, LabelLog, Dispatcher) \> Control-Flow Reconstruction
            SourceCodeLine[Round] = TotalLine
        end if
        OptimizedLine = -1, RoundOptimizedLine = 0
        while OptimizedLine == 0 do
            OptimizedLine = 0
            OptimizedLine, RoundOptimizedLine += DeadCodeElimination(OptimizedCode)
            OptimizedLine, RoundOptimizedLine += CopyPropagation(OptimizedCode)
            OptimizedLine, RoundOptimizedLine += ConstantPropagation(OptimizedCode)
            OptimizedLine, RoundOptimizedLine += ArrayValuePropagation(OptimizedCode)
            OptimizedLine, RoundOptimizedLine += ConstantFolding(OptimizedCode)
        end while
        SourceCodeLine[Round+1] = TotalLine - RoundOptimizedLine
        ObfuscatedSourceCode = OptimizedSourceCode
        OptimizedSourceCode = NULL
        Round += 1
    end while
    OptimizedSourceCode = ObfuscatedSourceCode
end procedure

As a result of our experiment, the control-flow graph became somewhat serialized. However, the functionality was not destroyed. After optimizing source codes used in the experiment, it was confirmed that our execution results were the same as those of the original source code. For example, in the case of CRC32, the correct value is calculated and output. Likewise, the other cases are in line with this.

In addition to the self-produced source code, we present the results of our experiment with optimization using open-source. The source code used is “Juliet Test Suite” provided
FIGURE 16. Flowchart of SCORE.

TABLE 2. SCORE results.

|        | Time (Seconds) | Before (Lines) | After (Lines) | Optimization Ratio (%) |
|--------|----------------|----------------|---------------|------------------------|
| HelloWorld | 0.1229          | 1,912          | 427           | 77.66                  |
| CRC32   | 0.8471          | 1,760          | 433           | 75.39                  |
| Base64  | 4.9617          | 5,008          | 1,632         | 67.41                  |
| Bitmap  | 4.8972          | 6,897          | 2,158         | 68.71                  |

by the National Institute of Science and Technology (NIST), which is a collection of test cases with 118 different Common Weakness Enumerations (CWEs).

Since the volume of actual source code is overwhelming, and there are some limits to present all control-flow graphs individually, an average source code volume, optimization ratio, and optimization ratio in binary are presented in Table 3. As a result of optimization, it should be confirmed that the program functionality is not destroyed. Thus, cases that could not immediately confirm the execution result were excluded from the experiment (e.g., infinite loop, network listening, etc.). Experiments were conducted on 30 types of vulnerable functions, and the average number of lines of each vulnerable function was about 50 lines.

As the source code including the vulnerability has a complicated internal structure, it was confirmed that the degree of optimization was greatly reduced than self-produced source code. However, this case can be improved somewhat if Pointer Propagation, which will be described later, can be further applied.

B. SCORE EVALUATION

In this subsection, we compare the ratio of SCORE optimization with other compiler optimization techniques to verify an additional effect that is obtained when the control-flow reconstruction method is applied. Since SCORE applies the source-to-source optimization mechanism in this paper, it is
TABLE 3. Results with test suite (Average value with 30 source codes).

| Time (Seconds) | Before (Lines) | After (Lines) | Optimization Ratio (%) |
|----------------|----------------|---------------|------------------------|
| 9.1229         | 3,412          | 1,436         | 57.91                  |

TABLE 4. SCORE evaluation (Size: (Bytes), assembly: (Lines)).

| Target Function | Size | Assembly | GCC (-O3) | Size | Assembly | MSVC (-O3) | Size | Assembly | ICC (-O3) | Size | Assembly | SCORE (-O0) | Size | Assembly |
|-----------------|------|----------|-----------|------|----------|------------|------|----------|-----------|------|----------|-------------|------|----------|
| HelloWorld      | 11,168 | 2,201 | 9,848 | 2,085 | 10,011 | 2,008 | 9,216 | 1,782 | 4,716 | 860 |
| CRC32           | 10,884 | 2,352 | 9,436 | 1,960 | 9,271 | 1,803 | 9,984 | 1,920 | 6,996 | 1,557 |
| Base64          | Compile Error | Compile Error | 29,087 | 5,346 | 24,648 | 4,867 | 6,392 | Compile Error | Compile Error |
| Bitmap          | Compile Error | Compile Error | 33,879 | 7,731 | 29,973 | 6,392 | 7,656 |

necessary to use the same source-to-source optimization solution to evaluate our results. However, unfortunately, optimization tools that perform C/C++ source-to-source optimization have not been studied or developed to the best of our knowledge. This is the reason that the source-to-binary optimization tools (i.e., C/C++ compilers) are included as comparison tools.

For the experiments, we selected Clang [21], GNU Compiler Collection (GCC), Microsoft Visual C++ (MSVC) [22], and Intel C++ Compiler [23]. Table 4 compares the binaries generated by each compiler and the compiled binaries after applying SCORE at the source code level.

Each compiler applies the -O3 option, which is the maximum optimization level in the compilation process. For the source code preprocessed by SCORE, the compilation process is performed using GCC but with the -O0 option, which is the minimum optimization option. SCORE provides source code optimization, and aside from a few cases, it shows a much higher degree of optimization at the binary level.

Unfortunately, SCORE is also found to inadvertently generate redundant code, particularly when the function to be optimized contained many lines of source code. This is because techniques such as loop unrolling are applied to perform serialization during the control-flow reconstruction process. Therefore, compared with other compiler optimization techniques, there may be cases where the amount of source code is larger. In addition, because of the obfuscation restrictions in SF itself, some compilers may not compile the obfuscated source code in certain situations.

Figure 17 shows the control-flow graph of the binary generated by each compiler. In the case of SCORE, because dispatcher removal is applied during the control-flow reconstruction process, the highly parallelized structure is eliminated. ICC is a compiler from Intel that provides a very high ratio of code and control-flow optimization, but even if the control-flow is serialized by ICC, it is still a big obstacle that hinders easy analysis. The main reason for this is that the dispatcher variables are fetched from outside the function to make the analysis of the dispatcher more challenging in SF. Therefore, it is obfuscated so that it can not grasp the control-flow statically as described previously.

During evaluation, we also found that certain compilers generated sporadic compile errors. We posit that this is because, during the source code obfuscation process, there may have been cases where some syntax was generated that could not be handled by the compiler in question, although it may have been handled well by the previous version of the compiler. For example, even in the case of the frequently used reserved word (e.g., __iob_func), an error occurs depending on the compiler type or version.

Source code obfuscation tool sellers do not know which version and what type of compiler is used for compilation. This suggests that it would be necessary to replace reserved words according to compiler type on a case-by-case basis. However, this is difficult to do for all compilers and all their respective versions. Still, this is the cause of the errors, and we note that a number of benign reserved words such as __iob_func are flagged in some cases.

Lastly, it is determined that the currently proposed Optimization Ratio can be improved slightly. In SF, the function that was obfuscated receives a large number of variables as function arguments (up to several tens as shown in Figure 10), and all variables are input in the form of void pointers.

If a module is designed to be able to extract and propagate the value of the corresponding pointer type (i.e., Pointer Propagation) by applying dynamic analysis, it is expected that the Optimization Ratio will increase to about 90%. This comprises the subject of our future research.

VIII. COUNTERMEASURES
SF obfuscate C/C++ source code with strong control-flow flattening. Flattened source code can be reconstructed if the analyst knows the order of code blocks, but it is difficult to analyze statically because of the additional obfuscation...
techniques applied in the flattened structure. Therefore, SCORE includes a module that performs the analysis dynamically with API insertion (i.e., label extraction API). By doing so, SCORE can extract the label list in the order of execution and reconstruct the source code.

Therefore, one countermeasure to SCORE could be to add an anti-tampering technique. In this section, we present the limitations of existing anti-tampering techniques and our experimental results of applying stand-alone, self-checkable source code.

A. ANTI-TAMPERING TECHNIQUES

Anti-tampering (i.e., tamper-resistant) techniques are used to detect and respond to the modification of objects. It is a type of software protection that can verify the integrity of a protected program. Normally, the tamper-resistant module consists of two parts: the tamper detection part, and the tamper response part.

Tamper resistance techniques are widely researched, but the bulk of these studies fall into one of the following categories.

1. An attester or verifier is required locally or remotely.

Simply put, tamper-resistant techniques require an attester or verifier for detection, as they detect tampering via computed values (e.g., hash value) from the original software. It can function in an external entity (e.g., the trusted server) or be embedded in the software itself.

2. A specific toolchain or module is required to insert the tamper-resistant technique.

In order to insert tamper-resistant techniques into the protected software, a specific toolchain or module may be required. For example, ASPIRE inserts Offline Code Guards to verify source code or binary, but it also needs toolchains to insert a verifier or attester.

3. If tamper-resistant software is distributed at the source code level, then the tamper-resistant part can be forced off somewhat easily.

Although most tamper-resistance software is distributed at the binary level, some is distributed at the source code level. In this case, it uses a Marker or Function form to verify the integrity of the target software, but it is usually exposed [5].

It is assumed that the additional countermeasure proposed in this paper is somewhat safe even if the stand-alone source code is stolen by a man-at-the-end (MATE) attacker. A secondary assumption is that there is no external entity, such as a remote attester or toolchain (1. & 2.). Also, if the tamper-detecting (i.e., verifier) part was overt, a MATE attacker could remove or modify the verifier with ease. Thus, it is necessary to blend the verifier with the obfuscated source code to make it difficult for an attacker to locate (3.).

B. SELF-CHECKABLE STEALTHY CODE ((SC)$^2$)

We present the Self-Checkable Stealthy Code Generator to complement the SF obfuscator with anti-tampering techniques.
techniques. The basic mechanism verifies the integrity of the process of compiled binary with a hash function because it is impossible for the source code to verify its own integrity. We implement a source code generator that can insert into the control flattened source code to calculate the hash value of select parts of the protected target function.

In the target function, the \((SC)^2\) is inserted in code block form, and it is located randomly. The hash value of the original target function (i.e., untampered) is also inserted into the source code. If the \((SC)^2\) inserted source code is compiled, then \((SC)^2\) checks whether the target function has been tampered or not before executing the target function. If the target function has tampered (i.e., hash value comparison fail), the process is terminated forcibly.

C. RESULTS & DISCUSSION
As shown in Figure 18, if tampered source code that is inserted the \((SC)^2\) is executed, the original functionality of the program is destroyed. To apply this module, less than 70 lines of source code are added, and execution time overhead increases by less than approximately 1 ms. Also, since the source code added in this process is written inline rather than with an API or user-specified function, it can be delayed to identify the functionality by API name or function name.

To bypass \((SC)^2\) or extract the execution order of code blocks without destroying original functionality, an attacker can use the following attacks.

- \((SC)^2\) Remove Attack
  An attacker can remove parts of \((SC)^2\) to avoid the check if they can distinguish which code blocks are part of \((SC)^2\) and which are part of the original functionality. However, \((SC)^2\) exists in a control-flattened form and shares the same dispatcher with the original functionality (i.e., it is not separated from the API or function form), so the attacker must analyze \((SC)^2\) code blocks, including the number (or position) of each code block. In addition, the index of the first code block that executes after \((SC)^2\) (i.e., the code block that would be executed first if \((SC)^2\) was not present) must be statically identified and rewritten to preserve original functionality. However, in the control-flattened source code, it is difficult to calculate the index of the code block executed after \((SC)^2\). Also, since the \((SC)^2\) generator additionally inserts into \((SC)^2\) code blocks parts of the original code to perform the original functionality (e.g., variable initialization), unconditional \((SC)^2\) code block removal without original code repositioning can destroy functionality.

- Dynamic Analysis Attack
  An attacker can dynamically analyze parts of \((SC)^2\) and original code blocks only to extract the execution order without tampering the source code (i.e., without using the API). However, this attack can be hindered by inserting additional anti-debugging code blocks to verify timeout between the execution time of \((SC)^2\) code blocks and original code blocks.

- Pre-Calculated Attack
  An attacker can bypass parts of \((SC)^2\) if the attacker can calculate the hash value of the target function. To do this, the attacker must ascertain the range of the protected part using \((SC)^2\) and the name of the hash algorithm (without the hash API name). However, since the hash value calculation part of \((SC)^2\) is obfuscated and

\[129494\]
control flattened, the time required for this analysis is significant.

As described above, when the (SC) module is inserted in code block form, malicious source code tampering can be prevented, as additional analysis time is required to remove or bypass it. An attacker can try to analyze and restore the control-flow with a dynamic analysis that does not tamper with the source code, but this can also be hampered by inserting anti-debugging code blocks or dummy code blocks.

However, since the integrity verification value (i.e., the hash value of the target function) varies depending on which compiler compiled the source code, this method has one limitation, in that the hash value must be stored for each compiler type. In other words, in order to have generality, a fixed value must be set as an integrity verification value regardless of compiler type. For example, it may be possible to utilize the number of API calls or opcode sequence extractions (with threshold) in the target function. This is a subject we intend to further explore in future research.

IX. CONCLUSION

This paper presented the design of SCORE, a source-level optimization tool, and its evaluation results. Code optimization is a well-known technique but is commonly applied at an IR level or lower. However, more research was necessary to improve readability by reconstructing the control-flow, as commercial obfuscation tools provide control-flow obfuscation at the source code level. In particular, as source-level, control-flow obfuscation is not optimized at all during compilation, analysis time of the compiled binary is greatly delayed. Furthermore, as the compiler strips information during compilation, undoing control-flow obfuscation at the binary level is more difficult than at the source code level.

SCORE performs control-flow reconstruction and optimization at the source code level, which greatly increases not only the readability of the source code but also the structure of the binary code. As optimization techniques are applied after control-flow reconstruction, more patterns can be optimized, providing a higher optimization effect than the optimization module of a compiler. In particular, this research is the first to demonstrate the necessity of source-level optimization and control-flow reconstruction empirically.

REFERENCES

[1] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” Dept. Comput. Sci., Univ. Auckland, Auckland, New Zealand, Tech. Rep. 148, 1997.

[2] I. You and K. Yin, “Malware obfuscation techniques: A brief survey,” in Proc. Int. Conf. Broadband, Wireless Comput., Commun., Appl., Nov. 2010, pp. 297–300.

[3] LCF-AT. Themida + WinLicense 2.X (Ultra Unpacker V1.4). Accessed: Jul. 13, 2020. [Online]. Available: http://en.pudn.com/Download/item/id/5440121.html

[4] LCF-AT. VMPProtect 1.x–2.x Ultra Unpacker V1.0. Accessed: Jul. 13, 2020. [Online]. Available: http://en.pudn.com/Download/item/id/2743288.html

[5] B. De Sutter, C. Basile, M. Ceccato, P. Falcari, M. Zunke, B. Wyseur, and J. d’Annoville, “The ASPIRE framework for software protection,” in Proc. ACM Workshop Softw. PROtection (SPRO), 2016, pp. 91–92.

[6] M. Vasudevan, “An architecture of class loader system in Java bytecode obfuscation,” Indian J. Sci. Technol., vol. 8, no. S2, pp. 291–294, 2015.

[7] A. Viticchie, L. Regano, M. Torchiano, C. Basile, M. Ceccato, P. Tonella, and R. Tiella, “Assessment of source code obfuscation techniques,” in Proc. IEEE 16th Int. Work. Conf. Source Code Anal. Manipulation (SCAM), Oct. 2016, pp. 11–20.

[8] F. Wang, J. Davidson, J. Hill, and J. Knight, “Protection of software-based survivability mechanisms,” in Proc. Int. Conf. Dependable Syst. Netw., 2001, pp. 193–202.

[9] M. Madou, B. Anckaert, B. De Bus, K. De Bosschere, J. Cappaert, and B. Preneel, “On the effectiveness of source code transformations for binary obfuscation,” in Proc. Int. Conf. Syst. Eng. Res. Pract. (SERP), London, U.K.: CSREA Press, 2006, pp. 527–533.

[10] C. Collberg, S. Martin, J. Myers, and B. Zimmerman, “The tigress diversifying $C$ virtualizer,” 2015.

[11] Compiler. Accessed: Jul. 13, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Compiler

[12] Semantic Designs. Accessed: Jul. 13, 2020. [Online]. Available: http://www.semanticdesigns.com/

[13] Stunnix. Accessed: Jul. 13, 2020. [Online]. Available: http://stunnix.com/

[14] M. Ceccato, M. Di Penta, J. Nagra, P. Falcari, F. Ricca, M. Torchiano, and P. Tonella, “The effectiveness of source code obfuscation: An experimental assessment,” in Proc. IEEE 17th Int. Conf. Program Comprehension, May 2009, pp. 178–187.

[15] StarForce. Accessed: Jul. 13, 2020. [Online]. Available: http://www.starforce.com/

[16] T. László and Á. Kiss, “Obfuscating C++ programs via control flow flattening,” Annales Univ. Scientiarum Budapestinensis de Rolando Eötvös Nominate, Sectio Computatorica, vol. 30, no. 1, pp. 3–19, 2009.

[17] S. Rolles, “Unpacking virtualization obfuscators,” in Proc. 3rd USENIX Workshop Offensive Technol. (WOOT), 2009, pp. 1–7.

[18] S. Banescu, C. Collberg, and A. Pretschner, “Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning,” in Proc. 26th USENIX Secur. Symp. (USENIX Secur.), 2017, pp. 661–678.

[19] M. Kobayashi, “Dynamic characteristics of loops,” IEEE Trans. Comput., vol. C-33, no. 2, pp. 125–132, Feb. 1984.

[20] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, “Compiler techniques for code compaction,” ACM Trans. Program. Lang. Syst., vol. 22, no. 2, pp. 378–415, Mar. 2000.

[21] C. Lattner, “LLVM and Clang: Next generation compiler technology,” in Proc. BSD Conf., vol. 5, May 2008, pp. 1–33.

[22] MSDN Help for Microsoft VC++ Compiler, Microsoft, Redmond, WA, USA, 2018. [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/reference/compiling-a-c-cpp-program?view=vs-2019

[23] Intel C++ Compiler User Guide, Intel, Santa Clara, CA, USA, 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top.html

[24] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, “Information hiding in software with mixed Boolean-arithmetic transforms,” in Proc. Int. Workshop Inf. Secur. Appl. Berlin, Germany: Springer, 2007.

[25] B. Yadegari and S. Debray, “Symbolic execution of obfuscated code,” in Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur. (CCS), 2015, pp. 732–744.

[26] S. Bardin, R. David, and J.-Y. Marion, “Backward-bounded DSE: Targeting infeasibility questions on obfuscated codes,” in Proc. IEEE Symc. Secur. Privacy (SP), May 2017, pp. 633–651.

[27] N. Eyrolles, L. Goubin, and M. Videau, “Defeating MBA-based obfuscation,” in Proc. ACM Workshop Softw. PROtection (SPRO), 2016, pp. 27–38.

[28] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, “Syntia: Synthesizing the semantics of obfuscated code,” in Proc. 26th USENIX Secur. Symp. (USENIX Secur.), 2017, pp. 643–659.

[29] J. H. Suk, J.-Y. Lee, H. Jin, I. S. Kim, and D. H. Lee, “UnThemida: Commercial obfuscation technique analysis with a fully obfuscated program,” Softw., Pract. Exper., vol. 48, no. 12, pp. 2331–2349, Dec. 2018.

[30] P. Garba and M. Favaro, “SATURN–Software deobfuscation framework based on LLVM,” in Proc. 3rd ACM Workshop Softw. Protection (SPRO), 2016, pp. 27–38.

[31] M. Olivieri, S. Bardin, R. Bonichon, and J.-Y. Marion, “Obfuscation: Where are we in anti-DSE protections? (a first attempt),” in Proc. 9th Workshop Softw. Secur. Protection, Reverse Eng. (SSPREW), 2019, pp. 1–8.

[32] M. Tuhkander, S. Islam, and P. Falcari, “Analysis of obfuscated code with program slicing,” in Proc. Int. Conf. Cyber Secur. Protection Digit. Services (Cyber Secur.), Jun. 2019, pp. 1–3.
JAE HYUK SUK received the B.S. degree in electrical and computer engineering from the University of Seoul, Seoul, South Korea, in 2012, and the M.S. degree in information security from Korea University, Seoul, in 2014, where he is currently pursuing the Ph.D. degree in information security with the Graduate School of Information Security. His research interests include software protection, program obfuscation, program deobfuscation, reverse engineering, and malware analysis.

YOUNG BI LEE received the B.S. degree in information security engineering from Soonchunhyang University, Asan, South Korea, in 2019. He is currently pursuing the M.S. degree in information security with the Graduate School of Information Security, Korea University. His research interests include software protection, program obfuscation, program deobfuscation, reverse engineering, malware analysis, and digital forensic.

DONG HOON LEE (Member, IEEE) received the B.S. degree from Korea University, Seoul, South Korea, in 1985, and the M.S. and Ph.D. degrees in computer science from The University of Oklahoma, Norman, OK, USA, in 1988 and 1992, respectively. Since 1993, he has been with the Faculty of Computer Science and Information Security, Korea University, where he is currently a Professor with the Graduate School of Information Security. His research interests include cryptographic protocol, applied cryptography, functional encryption, software protection, mobile security, vehicle security, and ubiquitous sensor network security.

***