Binary Program Analysis With No Prior Knowledge of Analysis Target

3.1 Introduction

This article describes binary program analysis technology that is assumption-free in that it requires no prior knowledge of the analysis target, which is being developed at the IIJ Research Laboratory.

Program analysis is a set of techniques for analyzing how programs behave. It can be broadly divided into dynamic analysis, whereby the program is actually run and its behavior is observed, and static analysis, in which the program's structure is reviewed without executing it.

Examples of dynamic analysis include unit tests for checking the integrity of program features, and fuzzing tools that feed a program random input to test its behavior. Examples of static analysis include optimization analysis—which seeks to enhance runtime computational efficiency by eliminating unnecessary code, precomputing operations, and so on—and static type checking to ensure data type consistency so as to avoid runtime errors caused by the program handling data in unintended ways. Whether dynamic or static, these sorts of program analysis techniques are incorporated into integrated development environments (IDEs), helping to streamline development and reduce bugs.

So these program analysis techniques are intended primarily for developers. Meanwhile, you may want to analyze the behavior of programs (binary programs) that are already in the wild. For instance, you may want to know how a suspected malware program behaves or check if firmware from a third party does anything suspicious. In such cases, the person seeking to analyze the program will not always have access to the source code or information on what compiler was used to create it. Dynamic analysis is still possible here. For example, quarantine systems that run suspected malware in a sandboxed environment to analyze its behavior are used in practice. But a problem with dynamic analysis is that only the control path actually executed can be analyzed. So dynamic analysis can be difficult in the case of anti-analysis malware that alters its behavior depending on the execution environment or firmware that has a backdoor enabling it to change its behavior according to specific input.

So to comprehensively analyze the behavior of binary programs, we need to perform static analysis in addition to dynamic analysis. The difficulty involved in the static analysis of binary programs can depend on how much prior knowledge you have about how the program was created. For example, if you can deduce what compiler was used, it may be possible to reconstruct the original source code based on the code patterns the compiler produces. This technique is called decompilation. If you can decompile a program, you can then use existing static analysis techniques on the reconstituted source code to analyze the program's behavior.

That said, you will not always have such prior knowledge available, or know whether you can trust it if it is available. The IIJ Research Laboratory is developing static analysis technology that can be applied to binary programs about which you have almost no prior knowledge, or in other words, when the program's origins are an enigma.

In the following sections, we discuss the difficulty of binary program static analysis and why the difficulty can increase depending on whether you have prior knowledge.

3.2 Binary Program Analysis

T.A program written in a language such as C/C++ (the source code) is converted by software called a compiler into a series of simple machine instructions a CPU can execute. This sequence of machine instructions is encoded into byte data according to the encoding method specified for the CPU architecture. A program expressed as a string of byte data like this is called binary code.

The binary code is embedded in a file with a specific format that makes it an executable file. In addition to the binary code, the executable also contains metadata specifying where to position the program in memory at runtime, where in the program to start execution from (the entry address), and what external library functions are called during execution. In this article, we treat these sorts of executables as binary programs. Static analysis of binary programs in the absence of source code is called binary code analysis.

A familiar example is standard virus detection software, which looks for malware signatures in binary code using a database of such signatures—distinctive byte data strings extracted from known malware. Systems that use machine learning on other metadata contained in executable files to detect unknown malware have also appeared in recent years.

Methods of analyzing binary code as mere byte data like this are suitable for automatic program classification applications such as malware detection, but to learn about how a program will behave in detail, you need to analyze it as a program and not just as a sequence of data. In this article, we call this binary program analysis.

With binary program analysis, you need to extract and reconstruct the program control structure from the binary code. A disassembler (Figure 1) is the first step. As mentioned earlier, machine instructions are converted into byte data according to the rules for each architecture. Doing this in reverse—converting byte data back into machine instructions—is called disassembly.

Simple disassemblers use linear sweeping, which involves disassembling instructions in sequence from the beginning of the binary code. If it encounters non-program data, a linear sweeping disassembler may not be able to correctly disassemble the program from that point onward. Advanced assemblers like IDA Pro, on the other hand, use recursive descent to recursively follow direct jump instructions (a program control instruction for which the destination address is specified as part of the instruction) starting from the entry address (Figure 2). Once a recursive descent disassembler encounters an indirect jump instruction (control instruction for which the destination address is stored in a register or memory), it can proceed no further. This is because, in general, the destination address of an indirect jump instruction presents an undecidable problem for static analysis, so it is, in theory, difficult to proceed.

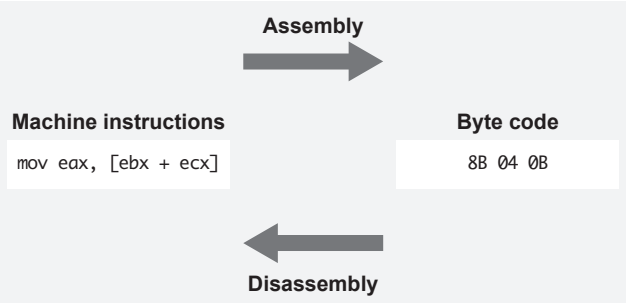

Figure 1: Disassembler

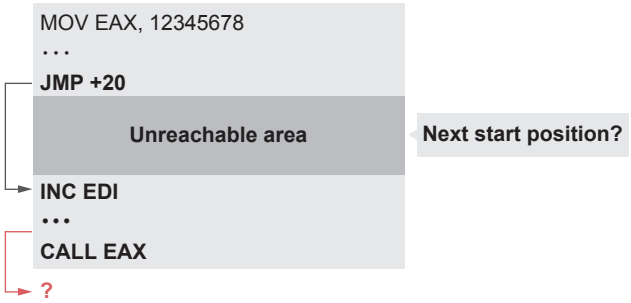

Figure 2: Recursive Descent Disassembler

Recursive descent, therefore, will result in unreachable areas—areas that cannot be reached by direct jump instructions alone. Disassemblers like IDA Pro use various heuristics to identify potential starting positions within these unreachable areas and restart the recursive-descent process. One such heuristic is a method is function identification.

Program development generally involves breaking the program logic into separate functional units, referred to as functions or procedures, to enhance code reusability and development efficiency. How parameters are passed to functions, and the way in which return values (the result of a function's calculations) are received, is determined according to the calling conventions of the CPU, operating system, etc. When functions are compiled, the compiler inserts the necessary preamble and post-amble code according to the calling conventions. This processing produces specific patterns depending on the compiler, so by finding these patterns, you can infer the location of functions. This type of analysis is called function identification.

The entire program can be disassembled by using function identification to break a program into functions and then recursively descending through each of them (Figure 3).

The use of function identification to determine the starting position of functions is premised on the assumption that the program was compiled according to the calling conventions. If this assumption does not hold, the disassembler will not be able to correctly reconstruct the program structure. It has also been reported that identifying the position of functions can be problematic even with programs generated by a compiler if the preprocessing and post-processing code patterns have been omitted due to heavy optimization[*1].

Indirect jump instructions are one reason it is not possible to recreate program structures using disassemblers alone. Using static data analysis to statically resolve indirect jump instruction destination addresses to the extent possible is called control flow reconstruction. As discussed, attempts to statically resolve the destination of indirect jump instructions run up against an undecidable problem, so a complete solution is not possible. Previous research such as CodeSurfer/x86[*2] and Jakstab[*3] has used abstract interpretation to seek approximate solutions to the destination address problem.

There is one more difficulty with control flow reconstruction, however. As discussed, programs consist of several functions. If function identification is first performed to divide a program into separate functions, intra-procedural program analysis, which analyzes each function independently, can be used. If you have insufficient prior knowledge to perform function identification accurately, you will need to use whole program analysis. Even if function positions cannot be identified in advance, programs are actually divided into a number of functions. So with whole program analysis, context-dependency must be taken into account.

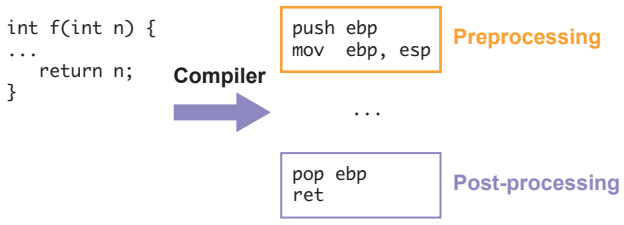

Figure 3: Preprocessing and Post-processing of Functions

*1 Andriesse2016: Andriesse, Dennis, et al. "An in-depth analysis of disassembly on full-scale x86/x64 binaries" 25th {USENIX} Security Symposium ({USENIX} Security 16). 2016.

*2 Balakrishnan2005: Balakrishnan, G., Gruian, R., Reps, T., & Teitelbaum, T. (2005, April). CodeSurfer/x86—A platform for analyzing x86 executables. In International Conference on Compiler Construction (pp. 250-254). Springer, Berlin, Heidelberg.

*3 Kinder2008: Kinder, J., & Veith, H. (2008, July). Jakstab: A static analysis platform for binaries. In International Conference on Computer Aided Verification (pp. 423-427). Springer, Berlin, Heidelberg.

For example, if a function is called from multiple program locations, control is transferred from each function call to the function, and once the function's activity is complete, control returns to the return point. After control has returned, if you want to refer to what the program state was before the function call, the call and return paths must match. This dependence of the analysis on paths is called context-dependency. If this context-dependency is not accounted for, distinguishing between multiple calls is not possible, so irrelevant contextual information muddies the mix when program state is being analyzed after a function call returns, significantly reducing analysis accuracy. Additional processing is needed to resolve context-dependency, such as the use of the stack to ensure consistent handling of function calls and returns. This sort of analysis is called inter-procedural program analysis.

So to enable highly accurate binary program analysis when function positions cannot be identified, you need to estimate function call/return positions during the analysis process. With existing static analysis methods, function positions are inferred by assuming that minimum calling conventions are followed (on Intel x86 architecture, for example, the CALL instruction is used to call functions and the RET instruction is used for returns).

Even the minimum calling conventions, however, cannot be guaranteed if you have no prior knowledge of the target program. For instance, the CALL/RET instructions may be used for purposes other than function calls/returns, or conversely, they could be replaced by other instructions. So to apply existing static analysis methods to binary program analysis, you need to have prior knowledge that guarantees the "goodness" of the target program.

The difficulties in binary program analysis discussed so far can be summarized as follows.

1. Disassemblers do not know the destination of indirect jump instructions.
2. In order to determine the destination of indirect jump instructions using existing static analysis methods, the program must first be divided into functions.
3. To identify the location of functions in a binary program, you need to make assumptions about, e.g., what sort of compiler was used and whether calling conventions are followed.

As such, existing binary program analysis methods require that you have prior knowledge about the conditions under which the program being analyzed was created, and that this knowledge is reliable.

Our method[*4] uses analysis of an intermediate representation that we propose to identify parts of a program as functions during the control flow reconstruction process, the aim being to make static program analysis applicable even in the absence of prior knowledge (i.e., even if the program is "bad").

*4 Izumida2018: Izumida, T., Mori, A., & Hashimoto, M. (2018, January). Context-Sensitive Flow Graph and Projective Single Assignment Form for Resolving Context-Dependency of Binary Code. In Proceedings of the 13th Workshop on Programming Languages and Analysis for Security (pp. 48-53).

3.3 Binary Program Analysis Using the Projective Single Assignment Form

In this section, we describe the method we are working on. As a working example, we use the 32-bit Intel x86 architecture program shown in Figure 4.

[Working example]

```
00401000:   xor ecx, ecx
00401002:   mov ebx, 0x40100c
00401007:   jmp 0x401017
0040100c:   mov ebx, 0x401016
00401011:   jmp 0x401017
00401016:   hlt

00401017:   inc ecx ; (A)
00401018:   jmp ebx
```

Figure 4: Working Example on 32-bit Intel x86 Architecture

In the example in Figure 4, the function code (A) is called twice, but instead of using the CALL/RET instructions, the code stores the return address in the EBX register and then jumps to (A), at which point it increments the ECX register by 1 and then jumps to the address stored in EBX, which takes it back to the instruction following the call. Existing analysis tools like IDA Pro cannot recognize code like this as function calls because it does not use the standard CALL/RET style (Figure 5).

In our research, we convert each machine instruction into simple assignment forms, and then further convert this into static single assignment (SSA) form. SSA is an internal representation format used in compiler optimization analysis. It changes variable names so that each variable definition is unique. This clarifies the definition-and-use (def-use) relationship of each variable, making it easy to understand the

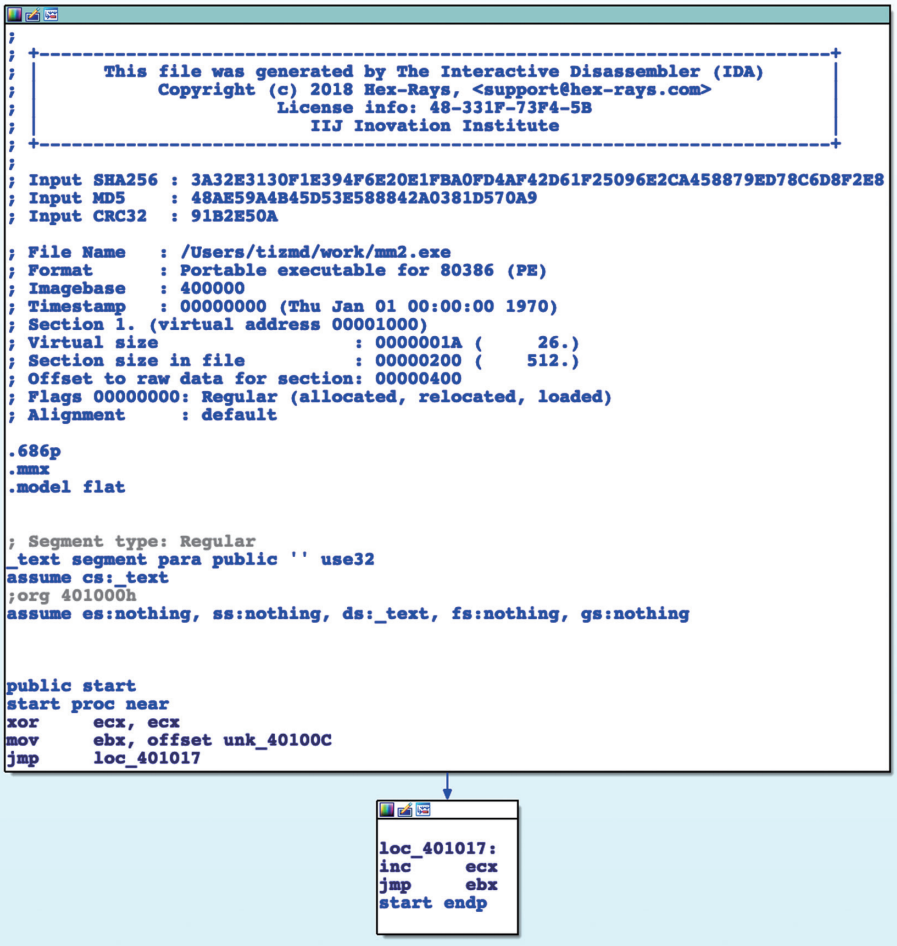

Figure 5: Example of Disassembly in IDA Pro

flow of information. For example, in Figure 6, the two ECX assignments are differentiated as ECX_1 and ECX_4.

Here, the code ends with an indirect jump to the return address stored in the EBX register (the jump instruction is expressed as an assignment to the program counter [EIP]). If you trace the definition of EBX_2 on the the right-hand side of the assignment, you can easily see that it is 0x40100c. So the destination address of this indirect jump expands to 0x40100c.

Figure 7 graphs the point up to where the second (A) call is completed. SSA expresses information merge points by introducing a pseudo-function called the Φ-function. For example, in the assignment statement $EBX_8 \leftarrow \Phi_8$ (3: EBX_2, 7: EBX_6), the EBX register values EBX_2 from node 3 and EBX_6 from node 7 merge and are newly assigned to the variable EBX_8. As before, if we trace the definition of EBX_8, it is expressed with a Φ-function as Φ_8 (3:0x40100c, 7:0x401016).

This means that if control reaches node 8 from node 3, the EBX register takes the value of 0x40100c, whereas if control comes from node 7, it takes the value 0x401016, so there is a merging of information. In our research, we refer to this changing of destination addresses due to information merging at certain points as context-dependency. Here, the code in the range from 0x401017 to 0x401018 is reused by multiple contexts, so we can infer that this is a function.

If context dependency is detected in this way, our method inserts a pseudo-function called the Π-function (Figure 8). The Π-function acts as a projection function for the Φ-function. For example, the expression $\Pi_{3 \to 8}$ (...) means that the information coming from node 3 is extracted from the information merged at node 8, so it is evaluated as $\Pi_{3 \to 8}(\Phi_8$ (3: X, 7: Y)) ⇒ X. This extension of the SSA form with Π projective functions is our own novel approach, which we call the projective single assignment (PSA) form.

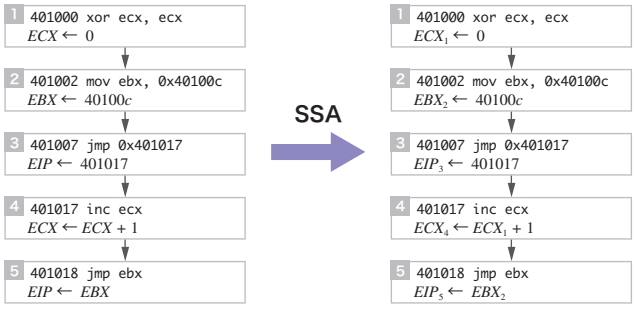

Figure 6: SSA Form: Up to the end of the first (A) call

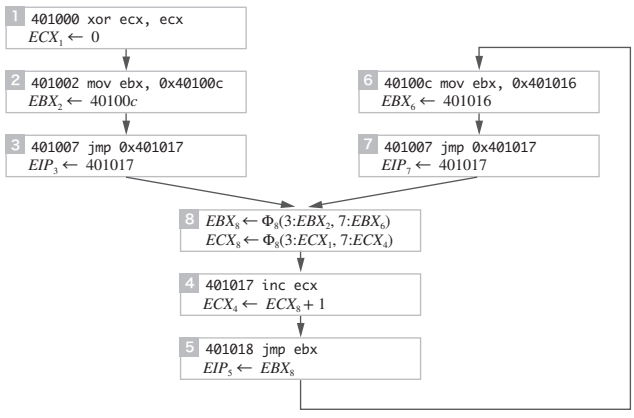

Figure 7: SSA Form: Up to the end of the second (A) call

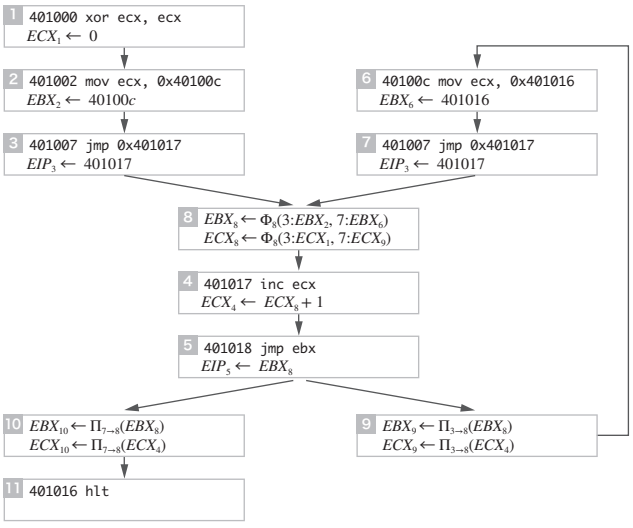

Figure 8: PSA Form: Insertion of Π-functions

Using Π-functions, the value of the ECX register when the program ends (node 11), for example, can be derived as follows.

$$ECX_{10} \Rightarrow \Pi_{7\rightarrow 8}(ECX_4) \Rightarrow \Pi_{7\rightarrow 8}(ECX_8) + 1 \Rightarrow \Pi_{7\rightarrow 8}(\Phi_8(3{:}ECX_1, 7{:}ECX_9)) + 1$$
$$\Rightarrow ECX_9 + 1 \Rightarrow \Pi_{3\rightarrow 8}(ECX_4) + 1 \Rightarrow \Pi_{3\rightarrow 8}(ECX_8) + 2 \Rightarrow \Pi_{3\rightarrow 8}(\Phi_8(3{:}ECX_1, 7{:}ECX_9)) + 2$$
$$\Rightarrow ECX_1 + 2 \Rightarrow 2$$

So by extracting context-dependency during the reconstruction process, we are able to resolve programs even if they do not follow calling conventions..

3.4 Application: Verifying Buffer-Overflow Safety

NThe projective single assignment adds not only projective Π-function but also conditional Γ-functions, which record the branching condition at each conditional branch. If loops are used within a function to rewrite data on the stack, the use of Γ-functions makes it possible to determine whether it is possible for the return address on the stack to be overwritten.

For example, converting the program in Figure 9 to PSA form and simplifying it results in Figure 10.

Here, Ld (M, A, N) denotes an N-byte value being read from address A in memory state M, and St(M,A,X,N) denotes the N-byte value X being set at address A in memory state M. In this example, when the function finishes, the program jumps to the 4-byte return address stored in the stack location expressed as $EBP_1 + 4$. The condition under which this area will be overwritten during the for loop is expressed in the PSA form as follows.

$$\Gamma(i_2 < 10, EBP_1 + 4 \leq EBP_1 - 10 + i_2 \leq EBP_1 + 8)$$

This means whether $EBP_1 + 4 \leq EBP_1 - 10 + i_2 \leq EBP_1 + 8$ is satisfied under the condition that $i_2 < 10$. Since i_2 is defined as Φ (i_1, i_4) and i_4 is defined in the loop, i_2 will change within the loop. If an i_2 value that satisfies this condition exists, there is a possibility of the return address value being overwritten during the loop. In this case here, however, it is easy to see that no such i_2 value exists. In other words, it is guaranteed that this loop will not change the return address area.

In practice, loop conditions and the memory overwrite conditions appear in more complicated forms. Finding loop invariants (conditions that do not change within a loop) is important in determining whether conditions such as these that vary within loops are satisfiable. Research on analysis methods for automatically evaluating such loop invariants has advanced in recent years and has been implemented in SMT solvers such as Z3. In our research, our objective is to extract loop conditions and memory overwrite conditions from binary programs and automatically calculate loop invariants using an SMT solver. This analysis will not always determine a solution within a specific timeframe, but if it can determine there to be no possibility of an overwrite, it

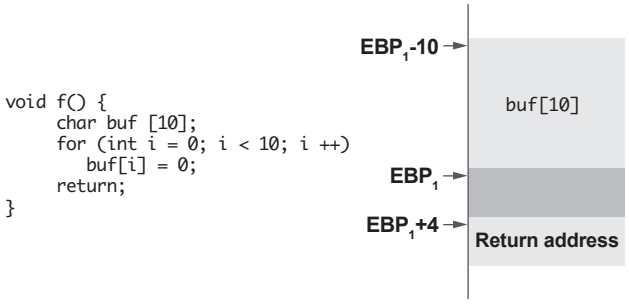

Figure 9: Example Program

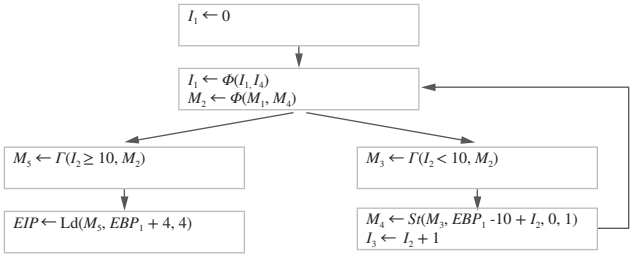

Figure 10: Loop

can guarantee that no buffer overflow will occur. And if it does identify the possibility of an overwrite, you can investigate the conditions under when overwrites can occur.

We are currently studying the application of this method to technology that detects vulnerabilities, such as buffer overflows, and the presence of backdoors, such as Trojan horses, in embedded firmware in AI edge devices and the like.

3.5 Conclusion

This article has described the assumption-free binary program analysis technology being developed at the IIJ Research Laboratory. Existing static analysis methods require that a program is first divided into separate functions, but doing this requires prior knowledge of or assumptions about how the program was generated. Using an extension of the SSA form, our method makes it possible to identify the location of functions by evaluating the destination of indirect jump instructions while also extracting context-dependency. This means that program analysis can be performed even on "bad" programs about which no prior knowledge can be obtained.

However, static binary program analysis involves undecidable problems, so no analysis method can provide a complete solution. Even with our method, we halt the evaluation and generate an approximate solution when the evaluated form becomes bloated and it appears that finding a static solution will be difficult.

Another method for binary program analysis is symbolic execution. This method of analysis sits somewhere between static and dynamic analysis. In 2016, mayhem[*5] and angr[*6], analysis tools that use symbolic execution, were among the leaders in the Cyber Grand Challenge, an IT security automation contest created by DARPA. Symbolic execution can detect if a program might produce dangerous states such as buffer overflows. But proving that a program is safe—meaning that it cannot produce any dangerous states at all—requires exhaustive execution, which is not something that symbolic execution is suited to. In this respect, we believe symbol execution and our method can complement each other.

Looking ahead, we aim to develop binary program analysis tools that integrate our method with other analysis techniques such as symbolic execution and dynamic analysis.

Acknowledgment
This research is carried out as part of "Research & Development on Fundamental Technologies Required for Comprehensive Security Evaluation of AI Edge Devices" work commissioned by Japan's New Energy and Industrial Technology Development Organization (NEDO).

Tomonori Izumida
Researcher, IIJ Innovation Institute (since 2015). PhD (information science).

*5 Cha2012: Cha, S. K., Avgerinos, T., Rebert, A., & Brumley, D. (2012, May). Unleashing mayhem on binary code. In 2012 IEEE Symposium on Security and Privacy (pp. 380-394). IEEE.

*6 Wang2017: Wang, F., & Shoshitaishvili, Y. (2017, September). Angr-the next generation of binary analysis. In 2017 IEEE Cybersecurity Development (SecDev) (pp. 8-9). IEEE.