The Effectiveness Of Bytecode Decompilation

Robert Enyedi
MA Students BRIE
renyedi@yahoo.com

Abstract

High-level bytecodes used by object-oriented managed execution environments make it easy to decompile them. This paper studies the reasons that make bytecode decompilers such efficient and presents basic obfuscation techniques as an efficient protection against binary code reverse engineering.

1. Introduction

The decompilation process is the reverse of the compilation process. Decompilation starts from the binary representation of a program and its finality is the source code that was the initial input for the compiler.
A decompiler is a program that reads a program expressed in a low-level programming language (usually machine code) and translates it to an equivalent program expressed in a high-level language (Figure 1).

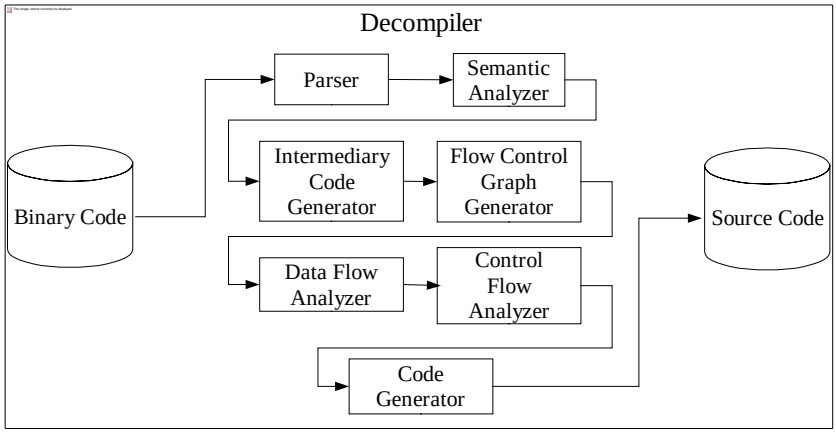

Figure 1The structure of a decompiler

For low-level machine code the decompilation process raises several theoretical and practical issues. Some of these can be solved by using heuristic techniques while others cannot be completely solved. Due to these limitation, a decompiler for low-level machine code accomplishes an automatic translation for some programs and a semi-automatic translation for other programs. Bytecodes represent high-level object-oriented machine code. Bytecode decompilers have fewer limitations, but there are obfuscation techniques to impose additional ones.

2. Decompilation Of Low-Level Binary Code

Decompilers for low-level binary languages face several issues due to the nature of the input code.
Recursive undecidability is an issue raised by low-level languages that mix data and instructions. Given a binary code, the separation between data and instructions is a partially computable problem and thus there is no splitting algorithm that can solve any situation.
Auto-morphing code refers to instructions or data that are modified during the execution of a program. This was a widely used technique in the 1960s and 1970s when computers didn't have too much memory and it was difficult to execute large programs. Even if today such limitations do not exist, the auto-morphing code technique still exists.

Idiomatic expressions are a sequence of instructions that form a logical entity and which together have a meaning that cannot be derived from the individual meanings of the instructions. Examples of idioms are bit shifting operations to simulate arithmetical operations and instructions groups for adding multi-byte variables.
Architectural dependent restrictions relate to the inner workings of a specific architecture. For instance, most of the modern computing platforms prefetch instructions while the microprocessor executes other instructions. This means that preloaded instructions are stored in a different place from those that are in the RAM. When a program uses auto-morphing code to change an instruction from the RAM, then, if the instruction has already been fetched, it is changed only in the RAM but not also in the buffer. This way only the prefetched code is executed. The analysis of this type of code needs to be done with the complete emulation of the executing machine.

3. Decompilers For Bytecodes

The construction of a decompiler faces several categories of problems when the input is a binary code expressed in a low-level machine language. The spread of object-oriented execution platforms (such as the Java Platform and the .NET Platform) produces an increasing base of bytecodes. Decompilers for these platforms are much more efficient in the reverse engineering of the original source code.
Bytecodes have such a high level object-oriented structuring that virtual machines generally offer a mechanism to perform reflection of binary code. Reflection is the process of analyzing the structure of the binary code during runtime. Using reflection it is possible to query for methods and fields of a class including typing and signature as well to perform dynamic class loading and execution. This is a powerful feature that allows writing of a much more runtime-aware software. Reflection is also a way to overcome several language limitations and to write more generic code (e.g. one can execute a method of an object with a certain signature, whatever the inheritance hierarchy of its class is).Given that bytecode data and code is object oriented there aren't mechanisms to perform such low-level static optimizations as mixing data with code. Actually there isn't any reason to allow static optimizations for bytecodes given that the execution is interpreted. The virtual machine performs a large set of dynamic optimizations of bytecodes which are more efficient. Because of the high structuring of bytecodes recursive undecibility and the von Neumann architectural limitations aren't issues for bytecodes.
Regarding auto-morphing, bytecode instructions do not have the ability to perform instruction modifications. Recently there have been evolutions of virtual machine runtimes to offer an instrumentation mechanism. Instrumentation is the ability to dynamically construct a class or a method. Even with a this kind of mechanism, it is the system runtime's ability to perform instrumentation and the decompiler can detect the use of standard runtime classes. What it can't do is the static decompilation of dynamically generated bytecodes.
Another characteristic of bytecodes is that instructions are of a much more higher level than low-level binary code. However, it is still possible to have idiomatic expressions just like in native binary code, although much simpler ones. One can still perform binary divisions and multiplications by 2 using shifting instructions.
Inlining of methods in order to speed up execution is a static optimization. Inlined methods also sacrifice reflection abilities. One of the main goals of virtual machine based execution is to automate as much routine tasks as possible. In traditional programming languages, there are two ways for a method/function to become inlined: explicitly and implicitly. Explicit inlining is done by the programmer using language constructs (usually a keyword) based on his of hers own abilities. Implicit inlining is done by the compiler. The compiler make the decision inline a method based on several factors including method size and number of references. Both of these ways are just best guesses, because only at runtime it is possible to tell how often a method gets executed and if it is feasible to make it inlined. In case of bytecodes, inlining is dynamic and is performed by the virtual machine at runtime. This way small methods that aren't used frequently or at all aren't inlined, but larger methods that are frequently called might very well get inlined. In any case, the decision is done automatically and there's not way to perform static inlining.

Bytecodes clearly define the boundaries of classes, methods and fields so that the structure of an application is maintained even after compilation. The linkage of external code or data is always dynamic so that a bytecode application doesn't include any system runtime code.

Limitations	*Low-level binary code*	*Bytecodes*
Recursive undecidability	Yes	No
von Neumann architecture	Yes	No
Code auto-morphing	Yes	Yes, but isolated
Idiomatic expressions	Yes	Yes, but simpler
Architectural dependent restrictions	Yes	No
Inlining	Yes	No
Runtime support code	Yes	No

Table 1Limitations of the decompilation process

A comparison of limitations for decompilation is presented in Table 1 for both low-level binary code and bytecodes. It clearly shows that bytecode decompilation faces less more limitations than low-level binary code.

Bytecode obfuscation techniques

The protection of intellectual property (IP) is a major issue for companies that develop software . In order to limit the ability and verbosity of decompiled bytecodes, there have been developed various obfuscation techniques. Some of these techniques prevent full decompilation, others do not prevent it but instead make the decompiled code difficult or impossible to understand.

Layout obfuscations are techniques of altering information that is unnecessary for correct program execution. Identifier scrambling is a common technique. It renames the declarations and usages of an identifier. It is a simple yet effective technique given that identifier names do have a semantic that are the primary way to understand the program. It should be noted that usage of this technique is limited in case of code that needs to be analyzed and manipulated by reflection. This is because identifier names become different or those from compilation time.

Another category of obfuscation techniques is that of data obfuscations. These include data storage obfuscation (changing the scope of data), data encoding obfuscations (changing the order of magnitude of values), data aggregation (alter the structuring of data) and data ordering (e.g. the use of a bijective function to access the indexes of a vector).

Control obfuscations disguise the control flow of an application and include control aggregation (forcing method inlining), control ordering (change execution order of statements), control computation (dead code insertion, use of unconditional jumps when the source language does not allow them, introduction of expression redundancies). There are also preventive transformation techniques that modify the bytecodes in order to exploit known weaknesses of existing relevant decompilers. These techniques aim to make the decompilers unable to perform decompilation.

Most of these techniques try to make the reverse engineered code hard to understand. However, some of them (unconditional jump insertions and preventive transformations) effectively prevent the ability to decompile bytecodes yet allowing the use of reflection to dynamically handle bytecodes.

Conclusions

Software decompilation faces several technical limitations. Bytecodes are much easier to decompile than traditional low-level binary code. With the recent widespread of high-level bytecodes this became an acute problem from the IP point of view. There have been efforts to develop specific obfuscation techniques. Obfuscators limit decompilers in their task, but still aren't able to impose as much limitations

as low-level binary code. Except IP issues, decompilers are useful to understand existing software and prevent code loss (a real problem in large companies).

Bibliography

[COTH97] Christian Collberg, Clark Thomborson, Douglas Low - A Taxonomy Of Obfuscating Transformations. Technical Report 148, Department Of Computer Science, University Of Auckland, New Zealand, July 1997 – http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html
[CRCI94] Cristina Cifuentes - Reverse Compilation Techniques, PhD thesis, Faculty of Information Technology, Queensland University of Technology, July 1994 – ftp://ftp.it.uq.edu.au/pub/CSM/dcc/decompilation_thesis.ps.gz
[J2SE04] *** - J2SE 1.5 .0 New Features – http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#instrument
[LIYE99] Tim Lindholm, Frank Yellin - The Java Virtual Machine Specification, Second Edition – http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html
[MSDN04] *** - The Microsoft Developer Network – msdn.microsoft.com