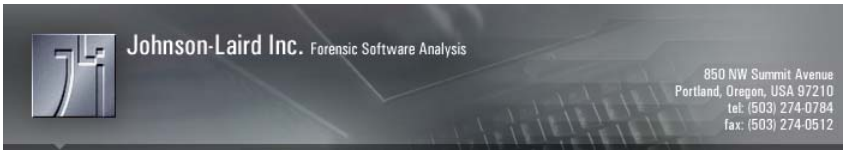
Reverse Engineering: Vulnerabilities and Solutions

By
Barbara Frederiksen-Cross (barb@jli.com) and
Susan Courtney (susan@jli.com)



ABSTRACT

The same characteristics that provide for cross-platform deployment of many modern software development languages also renders the software written in these languages extremely vulnerable to reverse engineering. At the same time, reverse engineering tools and techniques have become much more sophisticated. The convergence of these two developments creates substantial risk for software developers with respect to both the security of their software and protection of trade secrets and intellectual property that is embodied in the software.

Fortunately, the risks that reverse engineering poses to your intellectual property, competitive edge, and bottom line can be mitigated if you take proactive measures to protect your software against reverse engineering.

This paper first examines the ways in which software is vulnerable to reverse engineering and then explains techniques that may be incorporated into your software quality program to help protect your software assets against reverse engineering. The paper also discusses factors that must be considered and weighed when deciding which anti-reversing techniques to apply.

BIOGRAPHY

Barbara Frederiksen-Cross is the Senior Managing Consultant for Johnson-Laird, Inc., in Portland, Oregon.

Susan Courtney is a forensic software analyst and has worked as a consultant for Johnson-Laird, Inc.

1 Introduction

This paper examines the areas in which software is vulnerable to reverse engineering, and explains techniques that may be used to help protect your intellectual property against reverse engineering practices.

The idea for this paper was inspired by the needs of a client who experienced first-hand the reverse engineering of their software and its proprietary communication protocols. The software in question was vulnerable to automated reverse engineering techniques because it did not employ effective anti-debugging and anti-tampering measures. As a result of these weaknesses, competitors reverse engineered both the operation of the software and a proprietary encrypted communication protocol that was used to control the client's hardware products. The information was used to create competing products, permit copying of proprietary content, and preempt control of the client's devices.

2 Disclaimer

While it is possible to hinder reverse engineering in a number of different ways, no software technology can completely prevent a program from being reverse engineered. Whether reversers are ultimately successful in reverse engineering well-protected software depends ultimately on factors such as their skill, motivation, and the amount of time, money, and technology they are willing to devote to the effort.

Anti-reversing techniques are used to obstruct reversers by making the process of reverse engineering extremely difficult, slow, and expensive, thereby discouraging all but the most persistent, skilled, and well-funded reversers.

Multiple factors must be considered when deciding which anti-reversing techniques to apply. Every anti-reversing approach has an associated cost that must be measured in terms of its impact on system performance, code reliability, implementation ease, deployment considerations, maintenance requirements, product testing and production debugging issues. The selection and effectiveness of anti-reversing techniques must also consider details specific to the software and the operating environment that it uses.

3 Reverse Engineering Overview

Reverse engineering is the process of working with byte-code[1] or compiled code to gain an understanding of a program's logic and data. Programmers will recognize at least some of the reverse engineering techniques discussed in this paper because many reversing techniques use common debugging tools for a familiar purpose – to reveal the exact operation of the software in a controlled set of circumstances. In addition to common debugging tools, there are also a variety of special purpose reverse engineering tools that can help a reverser automate this process.

Programs written in certain languages are inherently more vulnerable to reverse engineering than others. With programming languages/frameworks such as Java and .NET, an intermediary product, often called byte-code, is shipped to the customer instead of compiled binary code. These frameworks provide for platform-independent code development by running the byte-code version of the software within a virtual machine environment that hides the details of the hardware upon which the software is running. The virtual machine manages the program's execution environment including interaction with the underlying hardware and operating system so that the program need not be aware of platform-specific differences. For the byte-code program to run on a particular computer, the virtual machine environment

[1] Byte-code is a generic term used to describe intermediary output. In Java, the intermediary output is known as Java bytecode. In .NET, the intermediary output is known as MSIL, which stands for "Microsoft Intermediate Language."

on that computer compiles the byte-code to the machine-readable instructions required for that specific hardware platform.

Byte-code based frameworks are more vulnerable to reverse engineering because byte-code based languages rely on an intermediary form of program instructions that are often very close in appearance to the original source code. A savvy programmer can inspect unprotected byte-code to reveal a program's inner workings, or use special software to recreate source code that can be used (illegally) as the basis for a competing product.

The development frameworks for languages such as Java and .NET also contain many pre-coded base classes that support common programming tasks such as user interface, network connectivity, mathematic functions, cryptography, data access, database connections, and services required for web-based applications. Programmers writing software in these frameworks build their applications using these base classes in combination with the code they create. Base classes are well-documented, and use of such building blocks (when visible in the byte-code or executable) provides additional clues that help the reverser understand the operation of the code. These clues may also help the reverser isolate specific functions of interest within the code.

Reverse engineering is most often accomplished with the aid of specialized software tools such as debuggers, disassemblers, and decompilers that permit the reverser to examine the sequence of instructions that comprise a computer program.

The tools and techniques used by reversers fall into two broad categories: dynamic analysis, which is analysis of the software as it is running, and static analysis, which is performed against a stored copy of the executable software. Thus, countermeasures used to prevent reverse engineering fall into two primary categories: techniques used to prevent dynamic analysis of the program while it is executing and techniques used to prevent static analysis of the binary code.

3.1 Dynamic Analysis Tools

3.1.1 Debuggers

The most common tool for dynamic analysis is called a debugger. Debuggers are designed to interact with programs and their runtime environments to capture information about unexpected behaviors ("bugs") that might be encountered during software testing. But debuggers also provide would-be reversers with the ability to inspect the memory, data, and files the program is using, and to allow the reverser to trace the sequence of operations within a program while it is running.

Debuggers allow a user to monitor and control the environment of the program as it executes. By using breakpoints[2] to pause the program's execution when specific criteria are met, control can be surrendered to the debugger software. At that time, the user can examine the contents of the program's environment, including data stored in the program's memory and the contents of files or file output areas. Debuggers also allow the user to step through the program one instruction at a time as the program is running ("single stepping")[3] and to capture and log various events and information as the program runs.

Most debuggers monitor the program from a very low-level perspective, capturing interactions between the program and the computer's processor, memory, and operating system. Program logic is generally represented in both machine language and assembly language formats. Even modestly complex programs may include millions of low-level machine or assembly language instructions, so

[2] Breakpoints can be set based on a variety of criteria, such as execution of specific instructions, access to specific memory locations, invocation of particular programming interfaces, or access to specific files.
[3] This process lets the reverser control program execution so that each individual instruction the program issues can be intercepted and examined.

reverse engineering an entire program start to finish by using a debugger is typically a tedious and time-consuming process.

A skilled reverser often reduces the volume of code that needs to be reverse engineered by exploiting her knowledge of the points at which an application program interfaces to the Windows operating system or to the user. By analyzing the program's use of interfaces, a reverser may be able to quickly locate the specific processing operations of interest within the targeted program. As a simple example, if the target program prompts the user to enter an activation key, and a reverser wishes to see the test required to validate the key, they can do so using debuggers such as Ollydbg, Shadow, WinDbg, or SoftIce. The reverser first searches for the error message sent to the user when an invalid key has been entered. This technique quickly localizes the area of the program that performs the processing of interest. By working backward from the point where the error message is issued the reverser can identify where in the code the activation key is evaluated, what tests are used to determine a valid key, or where to make modifications that will permit the key test to be bypassed entirely. By setting a breakpoint at the user prompt, the reverser can replay the prompt/response/validation code sequence as often as necessary to confirm her understanding.

Similarly, if a reverser can identify key data variables, file accesses, or memory locations, she can set breakpoints that are triggered by access to these locations, and use access or modification of these items as the starting point in her analysis. Once a breakpoint is reached, the program pauses and the reverser can inspect both the currently executing instructions and any associated data values in memory. This technique is often helpful to identify processing of proprietary data transformations, including encryption and decryption methods.

3.1.2 Dumpers

Dumpers are programs that typically capture the contents of a program's working memory at a specific point (or points) in time while it is running. This copy of memory is saved to disk for subsequent analysis. Most dumper programs also capture and save additional information that relates to the environment in which the program is running and information managed by the Windows operating system that concerns the program's operational state. Examples of memory dumpers include programs such as Explorer Suite and LordPE. Many dumpers also incorporate some of the dynamic capabilities common in debuggers.

By analyzing a memory dump, a reverser can examine the data the program was using at the time the dump was taken. Further, the reverser can also inspect all portions of the program that are present in memory at the time the memory dump is captured. The reverser can extract the memory values to a separate file for subsequent analysis. Once extracted, the reverser can use specialized software such as a disassembler to analyze and translate any portion of the program that is not encrypted, thereby rendering the captured memory contents as a series of human-readable instructions or data values.

3.1.3 Virtual Machines

Virtual machines ("VMs") are software implementations of real hardware. A VM imposes a layer of abstraction between the running program and the computer's hardware and operating system. This abstraction layer provides a controlled environment in which any program can be run as though on the real hardware, but under the watchful eye of the VM software. Virtual machines are sometimes used in the context of dynamic analysis so that a reverser can save periodic snapshots of the program state and operating environment as she attempts to reverse engineer the program. If the program terminates for some unexpected reason, these snapshots can be used to start the reverse engineering session again at some predetermined point, so that the reverser does not need to start over from the beginning of the run. The use of virtual machines can also help a reverser defeat some forms of anti-debugging protection.

3.1.4 Other Tools Used In Dynamic Analysis

Reversers also employ registry and file monitoring software during dynamic analysis. A registry monitor can be used to intercept registry keys that may be used to store application information. This tool facilitates the recovery of license keys or decryption keys, if they have been stored in the Windows registry.[4]

A file monitor can be used to trap accesses to all files used by the application. This analysis can help the reverser understand the program's operation. It also helps the reverser identify files that may be used to hide a security algorithm or encryption key so that this information may be intercepted.

3.2 Static Analysis Tools

3.2.1 Disassemblers and Decompilers

Disassemblers and decompilers are used to analyze a static copy of the program stored on disk. Generally speaking, a disassembler reads a machine-readable version of a program, parses it to identify the discreet sequences of instructions that make up the program, translates these instructions to a human-readable form, and writes the translation to a file for later analysis.

The output of a disassembler is usually low-level assembly language source code that provides a human-readable representation of the program at a much lower level of abstraction than the original source code.[5] Most debuggers include disassembly capabilities while other products, such as IDA Pro, are directed more specifically to the task of disassembly, and may include special capabilities to optimize the resultant translation.

Because the assembly code version of a program may be millions of lines long, including instructions for very common functions such as displaying windows and accepting user input, reversers seldom attempt to reverse engineer the totality of the program. Instead, the reverser searches the converted code for a particular character sequence (such as a message, literal value, or exported function name) to use as a starting point, and then works backward or forward from that point to locate desired algorithms or functions in the program. Once located, the reverser can analyze the specific sequence of instructions to understand how they operate.

Decompilers and reverse compilers are similar to disassemblers. However a decompiler reads in executable code, assembler code, or byte-code, and writes out source code in a higher level language, such as C or the original programming language used to write the program. With the exception of decompilers used for byte-code languages, the resultant code is often very different from the original source code, though much easier to read than low-level assembly language.

Products such as SourceAgain, JReversePro, JODE and Jdec produce good Java source code from .class files. Products such as Reflector and Reflexil perform similar functions for .NET code, allowing a reverser to extract very complete MSIL (.NET byte-code) that may be nearly identical to the original source code.

[4] The windows registry is a special directory that stores settings and options used for the Microsoft Windows operating system. It also contains information relating to hardware and user-specific settings. Many programs store information such as license keys and initialization settings in the registry.
[5] Note: This model does not hold true for development platforms such as Java and .NET. Disassemblers for these platforms read in byte-code and may produce output that is very similar to the original source code unless the byte-code has been obfuscated.

3.2.2 Other Tools Used In Static Analysis

Tools such as hexadecimal editors ("hex editors"), unpacker programs, and file analyzers are also useful when performing static reverse engineering. A hex editor allows the reverser to search and view a human-friendly representation of compiled programs. The hex editor may be used to run initial searches for textual elements, such as error messages, that can help identify portions of the program that are of particular interest. A hex editor may also be used to provide a human-readable view of binary code for visual inspection and analysis.

An unpacker program may be used to unpack compressed or encrypted code so that it can be processed with a disassembler or viewed with a hex editor. Many unpackers also include features that help to remove certain types of formulaic obfuscation.[6]

File analyzers are used to identify the compiler/packer/obfuscator that was used with the software of interest, so that the appropriate decompiler/unpacker/deobfuscator can be used.

4 Thwarting the Reversers

The primary techniques to prevent dynamic analysis include page guarding, debugger detection, and virtual machine detection. The primary techniques used to prevent static analysis are encryption and obfuscation. These techniques will be discussed respectively in this following section.

Techniques used to prevent static and dynamic analysis can be implemented directly in the program code as it is written, or they can be applied after the entire program has been written via the use of third-party software. In the latter case, the third party software reads in the source code or executable version of the software, performs a series of modifications to implement the desired protection techniques, and writes out the transformed source or executable. Applying protection after development is completed minimizes the impact on the development lifecycle.

4.1 Preventing Dynamic Analysis

Dynamic analysis is difficult to defeat because program instructions must be unencrypted to execute. This means that there will always be some portion of the program that is exposed to inspection by debugging tools or memory dumpers. Debuggers and memory dumpers can help defeat encryption by permitting the reverser to capture the portions of a program that are unencrypted in the computer's memory during execution. Once the areas of the program that handle encryption are identified, dynamic analysis can also be used to obtain decryption keys that are stored in memory and to locate and analyze decryption algorithms by tracing the program's instructions during execution.

4.1.1 Debugger Detection

To minimize a program's vulnerability to dynamic analysis, a program must first be able to detect that it is being run in a debugging environment, and then once detected, to take some form of defensive action. The tests for debuggers may be coded as a part of the program, or the protection may be applied via automated tools that provide a secure environment wherein the program runs. A general scenario for debugger protection is described in the following paragraphs.[7]

[6] Obfuscation is the generic name for a variety of techniques that are used to reduce a program's vulnerability to static analysis. Obfuscation techniques modify the program's symbols, internal data, layout, and logic in a way that preserves the function of the program but makes the program's operation much more difficult to decipher and understand.

[7] A complete discussion of debugger detection is beyond the scope of this paper. Detailed information is available online *at http://www.openrce.org/reference_library/anti_reversing* (last visited 2011-08-04)*;*

On startup, the code to detect debuggers loads first, and prevents the protected program from loading if it detects that a debugger is in use. If no debugger is present, the protected program is allowed to load and begin execution. As it runs, the protected program periodically relinquishes control back to the debugger detection software to ensure that the runtime environment is still safe. If a debugger is detected while the program is running, the debugger detection software initiates a pre-determined defensive action.

Possible defensive actions include immediately stopping the program, changing the behavior of the program so that the reverser is led to an erroneous understanding of its logic, and altering the program to force some eventual error after wasting the reverser's time.

4.1.2 Considerations for Debugger detection

Techniques to detect debuggers include inspecting specific debugger status information maintained by the operating system when a debugger is running, such as checking the names of running programs to identify any known debuggers, and creating a process that looks for evidence that debugger software is installed on the computer's hard disk.

Some debuggers create breakpoints by modifying the programs they monitor. These debuggers inject interrupts into the target program after it is loaded into memory. This activity can be detected by monitoring or scanning the program memory for these changes (a process called *check summing*). A related technique, called *page guarding*, seeks to interfere with the creation of breakpoints by preventing or healing such changes to the program.

Since the use of a debugger typically slows a program's execution, it is sometimes possible to detect the presence of a debugger by computing the amount of time required to run specific program instructions. Once computed, this value is tested against a predetermined threshold to see if the execution is slower than expected. This test is predicated on assumptions about how long the tested instruction should take to run in a particular environment, so this technique can only be used if environmental information is available before the program is shipped to customers.

The techniques used for debugger detection all require a solid understanding of both the operating system and the various debuggers available for the specific platform. Technical information required for debugger detection is subject to change with new releases of either the Windows operating system or new debugger software. Due to the specialized knowledge and skills required for debugger detection, the most pragmatic solution is often to add debugger protection via the use of commercial products, especially when this protection is needed for legacy software products.

One downside of debugger detection and defensive action is that the logic required to perform debugger checks uses system resources and therefore may affect performance. With certain techniques for debugger detection, it is possible that a user with a legitimate debugging session unrelated to the target program might trigger the defensive actions, causing the protected program to terminate unexpectedly.

To address these issues, most commercial products used to apply debugger detection allow the user to select among options that offer varying levels of protection.

4.2 Preventing Static Analysis

Encryption and obfuscation are the primary techniques used to prevent static analysis. These techniques may be used independently or combined together.

---

http://www.symantec.com/connect/articles/windows-anti-debug-reference (last visited 2011-08-04); or the four part anti-debugging series located at *http://www.veracode.com/blog/?s=anti-debugging* (last visited 2011-08-08).

4.2.1 Encryption

Encryption uses a mathematical formula to transform a program's byte-code or binary code into a form that is no longer comprehensible to the reverser or to automated disassembly software.

One approach that interferes with static reverse engineering is to use internal self-encryption of the program's binary code. A program that is self-encrypting is stored in encrypted form and only decrypted during execution. Using self-encryption protects a program because comprehensible instructions are only exposed to inspection during the comparatively short time when the program is actually running. This technique, sometimes called *packing*[8] or *binary encryption*, will prevent static analysis and automated disassembly, unless the reverser can find some way to first decrypt the program – i.e. through use of a specialized unpacker program or repeated memory dumps.

For encryption to be effective the target program and the decryption/encryption routines used to protect it must both be encrypted. In some cases, the decryption routine is designed to decrypt the entire program into memory at the start of execution. This form of encryption is extremely vulnerable since the entire unencrypted program can be read from memory (using a debugger or memory dumper) while the program is running.

Alternatively, a better technique called just-in-time encryption ("JIT encryption") decrypts only a small segment of the program as it is needed during execution, and then re-encrypts it before the next segment is decrypted.[9] This technique offers additional protection against attempts to capture an unencrypted version of the program as it runs.

So-called polymorphic encryptors may use multiple encryption algorithms and unpacking modules within the same program to render detection and reversal even more difficult.

4.2.2 Considerations for Encryption

Encryption techniques help protect against static analysis, but do not directly address dynamic analysis because instructions must be decrypted before they can be run. Encryption is most effective when only small portions of the program are decrypted into memory at any particular time. This technique prevents the use of dumper software to capture an unencrypted version of the entire program from memory.

Unfortunately many third-party packers and binary encryption products themselves have been reverse engineered. If a reverser can identify the specific packing/encryption tool used, an automated solution to assist with unpacking or decrypting the protected code may be readily available. To avoid this possibility, encryption techniques may be accomplished using custom software. This approach provides some additional security but at the expense of developing, deploying, and maintaining the custom encryption software.

Whether encryption software is purchased or custom written, there is always a potential down-side in terms of performance. From the user's perspective, software that is decrypted in its entirety at run time may appear to take longer to start. Software that uses JIT encryption will run slower than an

[8] Many varieties of packer software exist. Historically, packers were used to compress the size of executable code as an aid to distribution and storage on secondary media. A compressed executable includes the original program code, which has been compressed to reduce its size, packaged along with the relevant decompression code routines in a single executable. Running a compressed executable requires first loading the decompression routine to uncompress the original program code and then transferring control to the uncompressed code. Today, many packers include at least rudimentary capabilities to perform encryption and obfuscation.

[9] Just-in-time encryption should not be confused with just-in-time compilation, a different technique that is used to improve performance for programs written in bytecode-based languages such as Java.

unprotected version of the same software. How much slower is determined by the way that encryption is implemented and the specific processing performed by the decryption and encryption routines.

The impact of encryption on performance can be tuned to some extent by encrypting only selected portions of the overall software. The performance of JIT encryption is also sensitive to the size of the segment being decrypted, which can often be selected at the time encryption is added.

4.2.3 Obfuscation

In addition to encrypting, software may be obfuscated to help defeat static analysis. Obfuscation makes a program more difficult to understand and attempts to thwart automated disassembly tools.

In addition to transforming the program code to hide the original logic, an obfuscator may rearrange the internal organization of the software and add irrelevant code blocks ("garbage code") which serve to increase the apparent complexity of the program and lead reversers away from the true program logic. Some obfuscation techniques also introduce code changes that prevent automated disassemblers and decompilers from producing an accurate rendering of the program code.

Obfuscation is a generic name for techniques used to reduce a program's vulnerability to reverse engineering. Obfuscation techniques fall into three general categories: lexical transformations, control flow transformations, and data transformations. At best, obfuscation makes it time-consuming, though not impossible, to reverse engineer a program. The various types of obfuscation techniques, their effectiveness, and technical considerations are discussed below.

4.2.4 Lexical Transformations

Lexical transformations remove textual clues that reveal how the program operates making it harder for a reverser to *zero in* on specific algorithms and processing routines of interest. Lexical transformations can be used to replace program identifiers, such as the names used to identify program data, and function names that identify the program's processing routines. Lexical transformations may also be used to disguise textual content such as error messages or user instructions.

Lexical transformations are useful because they make the program harder to read and understand, but offer only a weak protection when used alone. Modern disassemblers include sophisticated tools such as call path analysis and visual representations of code structure. These tools are designed to help a reverser overcome the lack of meaningful names and symbols.

4.2.5 Control Flow Transformation

Control flow transformations seek to defeat disassemblers by exploiting weaknesses in the way the disassembler translates binary code into assembly language instructions. Control flow transformations may alter the order and flow of the program logic to make the logic harder to follow.

For example, some control flow transformation techniques add or modify conditional and branching instructions that complicate the program. In this type of modification the test results for the artificial branch condition must be calculated at the time the program is run, so a disassembler cannot know which branch would be taken and must therefore disassemble both. The obfuscator that transforms the software actually knows how the condition will be evaluated in advance, and structures the inserted branch so that when the program runs it will always take the same logic path that was followed in the original code.

Other control transformation techniques modify the actual "jump" instructions that control the logic flow in the binary code. In an unmodified program, the jump instruction often provides the address of the next instruction to be executed. To obfuscate this control flow, the program is modified to force it to compute the address for the target value of that branch. This technique prevents a disassembler from

determining the logic flow because a vital piece of data, the address of the next instruction, is not known at compile time.

Another technique used for obfuscation is to break up the logical order of a program, making it harder to follow. In this technique, high level code structures, such as functions, are replaced with less transparent abstractions. For example, a complex function such as processing a request from a user is broken into multiple components which are scattered throughout the program, perhaps interleaved with other code that serves some other purpose within a single routine. Once functional routines are broken up and dispersed, the control of the program is modified to ensure that these components are still executed in the proper order. This type of change can add significant complexity to the control flow but still preserves the original function of the program.

4.2.6 Data Transformation

Data transformations serve to disguise important data structures within the program by altering the way data is stored or represented within the program. Examples of techniques used for data transformation include modifying or splitting table structures, replacing literal values stored in the program with computations that deliver the same value, encrypting textual error messages and literal values, and dividing important structures into multiple pieces that are not all stored in the same way.

4.2.7 Combined Techniques

Many of the techniques described above may be combined to further strengthen their effectiveness. For example, the human-readable code may be compiled normally, and then further processed to replace all unconditional branches with conditional branches plus false predicates. Computations are inserted for “true” branch targets, to prevent detection of the true target by the disassembler. Targets of the false branches are pointed to positions before the real target, which are padded with bad or misleading code.

4.2.8 Automating Obfuscation

Obfuscation transformations may be applied to software using either manual or automated processes and to either the source code or the binary translation that is produced by a compiler. Performing manual obfuscation transformations can be very time consuming, thus obfuscation is commonly made using obfuscator software. There are many different obfuscator programs available in the marketplace, both as commercial products and as shareware. Most offer a variety of protective transformations, and give the user some level of control over what specific techniques are to be applied.

In a typical scenario, the target program is compiled, and then the executable binary code is input into the obfuscator along with parameters that tell the obfuscator which types of obfuscation are desired. The obfuscator transforms the code and writes out the obfuscated version of the software that will be made available to customers.

4.3 Considerations for Obfuscation

Obfuscation is most effective when combined with encryption, and it is normally best practice to use a combination of obfuscation techniques so that lexical content, control flow, and data structures are all transformed or obscured. The benefits of automated obfuscators include ease of use, ongoing support by a third party whose focus is automated obfuscation, and minimal impact on code development and maintenance. One drawback: most obfuscation techniques provide protection against static analysis, but do not directly address dynamic analysis. Obfuscation benefits must be weighed against their effect on system performance.

Transformations of lexical content and data structures are primarily directed to making a program harder to understand for a human reverser. Transformation of lexical content does not normally carry any performance penalty but it provides only modest protection against reverse engineering. Transformation of data structures usually has a comparatively small effect on performance. While it provides a greater hurdle than simple lexical transformation, it does not prevent the use of automated disassembly tools.

In contrast, transformation of logic flow can be very effective in defeating the automated tools used for static analysis, but these transformations may significantly affect performance. Depending on the specific techniques used, logic flow transformations can also make the program much harder for a human reverser to understand.

In addition to merely increasing program complexity, it is important to choose an obfuscator that produces transformations resilient to automated removal. Obfuscation transformations may increase the size of the program, its memory requirements, and its demand for processing resource.

Further, reversers understand that software developers may employ obfuscation tactics. Because many obfuscation transformations are somewhat formulaic, reversers may use deobfuscator programs[10] to attempt to restore the programs original structure. As the name implies, deobfuscator software is designed to help reversers analyze an obfuscated program by removing encryption and obfuscation transformations. The deobfuscator does so by attempting to detect and reverse the formulaic transformations applied by obfuscators.

5 Conclusion and Recommendations

None of the techniques used to protect against reverse engineering can guarantee that software will never be reverse engineered. That said, there are anti-reversing techniques that can make the task of reverse engineering more difficult and more time consuming. These techniques can be applied in one of two ways: they can be added as modifications to the program source code, or they can be applied to the executable version of the program before it ships to customers.

All of the anti-reversing techniques discussed in this paper can be added to a body of software via the use of automated software tools. These anti-reversing techniques may be applied to the entirety of the software or to selected portions. As a minimum, we recommend that the following protections:

- Encrypt the executable version of the software, preferably with JIT encryption
- Use a combination of obfuscation techniques to provide the software with additional protection against reverse engineering
- Don't forget to extend the same protection to software or firmware upgrades

A variety of products are available to automate the addition of anti-reversing techniques. These tools vary in sophistication, cost, techniques employed, ease of use, and effectiveness. The selection of automation tools should be predicated on an understanding of which techniques provide a best fit for your security goals, performance requirements, and cost sensitivity. A comparison of several of the most common products is included in Appendix A.

Finally, you may want to consider enhancing your legal protection to help you in the event your software is compromised. Although these measures cannot prevent reverse engineering, they can provide a tactical advantage to help you seek remedies in the event a breach does occur. Some of the options available to help you in this regard include strengthening the terms of your end user license

[10] Many common obfuscators have themselves been reverse engineered, revealing their specific techniques, algorithms, and encryption strategies. This information has been used by reversers to create automated deobfuscation software that can be used to attempt to reverse the effects of obfuscation. Such deobfuscators are also referred to as "unpackers."

agreement, filing a deposit copy of each version of your software with the US Copyright Office, and obtaining patent protection for any novel aspects of your product.

6 Appendix A: Feature Comparison

This chart summarizes feature and price information for a sample of nine alternate software protection products (including two different Dotfuscator versions). This sample is by no means exhaustive, but rather reflects the capabilities of some of the more popular or more robust offerings in this market.

Pricing key: $ cost range < $1000, $$ cost range $10K -$20K, $$$ cost range $50K-$100K+

	Dotfuscator Community Edition	**Dotfuscator Professional Edition**	**Cloakware Security Suite**
Supported Languages	.NET	.NET	Most compiled languages, no .NET
Encryption	N	some	Y
JIT Decrypt	N	N	Y - whitebox
HW locking	N	N	
Lexical Obfuscation	some	Y	Y
Control Flow Obfuscation	N	Y	Y
Data Obfuscation	N	Y	Y
Debugger Detection	N	some	Y
VM detection	N	N	?
Page Guarding	N	Y (SW, via checksum + custom method)	Y
Tamper checking	N	Y	Y
Integration Consulting	N	Y	Y
Licensing Model	Bundled with MS Visual Studio	Base +Per Developer	Annual license or per device
Cost	Free	$$	$$$
Hooks	Post CIL	Post CIL	Source and post compile
Time to deploy	Days	Days to 2 weeks	Est. 2 months
Notes	Not recommended as this offers only weak protection	Has recently added instrumentation platform which includes use analysis	

	StrongBit ExeCryptor	9Rays Spices Obfuscator	Silicon Realms Armadillo (Passport)	Oreans Themida & Code Virtualizer
Supported Languages	Most compiled, no .NET	.NET and managed C++ assemblies	Most compiled languages, no .NET	Most
Encryption	Y	String only	Y	Y
JIT Decrypt	Y - see note	N		Y – see note
HW locking	Y		Y	?
Lexical Obfuscation	?	Y	Via encryption	Y
Control Flow Obfuscation	Y - code morphing	Y	Via nanomites	Y – code morphing
Data Obfuscation	?	Y	N	?
Debugger Detection	Y	N	Y	Y
VM detection	?	N	N	Y
Page Guarding	?	N	Y	Y
Tamper checking	?	Y	Y	Ring 0
Integration Consulting	?	?	Custom build available at no charge	?
Licensing Model	?	Per build machine or enterprise	Per developer	Annual license
Cost	$	$$	$	$
Hooks	Post compile	Post CIL	Post compile	Post compile
Time to deploy	Days	Days to 2 weeks	Days	Days to 2 weeks
Notes	Very unique approach to obfuscation. Protected SW morphed and portions run in embedded VM. Partners with 9Rays	Partners with StrongBit	Custom build is recommended, offered at no additional charge	Code translated and executed via embedded emulator. Unpopular with Reverse Engineers

	Arxan TransfornIT	v.i.Laboratory CodeArmor
Supported Languages	Most compiled languages, lighter coverage for .NET (no repair guard)	Most compiled languages, separate version of product for .NET
Encryption	Y	Y
JIT Decrypt	Y	Y
HW locking	Y	
Lexical Obfuscation	Y	Via encryption
Control Flow Obfuscation	Y	N
Data Obfuscation	Y	N
Debugger Detection	Y	Y
VM detection	Y	Y
Page Guarding	Y – multi level w/ guards and repair guard	Y
Tamper checking	Y	?
Integration Consulting	Y	Y
Licensing Model	Perpetual, Tiered, or per app	Tiered, per app
Cost	$$$	$$ - $$$
Hooks	Post compile	Post compile/Post CIL
Time to deploy	Days to 6 wks	Days
Notes	At least 3-5% overhead, this assumes guards tuned	At least 2-4% overhead