

# **An Intellectual Property Threat Analysis of a .NET Windows Application**

*By Paul Mason*



## **Contents**

<b>1. Abstract</b> .....	3
<b>2. Introduction</b> .....	3
<b>3. A Brief Glance at Microsoft .NET Files</b> .....	4
<b>4. Description of PenMarked</b> .....	5
<b>5. PenMarked Intellectual Property Threat Analysis</b> .....	5
<b>6. Tamper Proofing as a solution</b> .....	7
<b>6.1 Overview</b> .....	7
<b>6.2 The Verdict</b> .....	8
<b>6.2.1 Dynamic Self-Checking</b> .....	9
<b>6.2.2 AOP Dynamic Self-Checking</b> .....	9
<b>6.2.3 Assembly Encryption</b> .....	10
<b>7. Discussion</b> .....	10
<b>8. Conclusion</b> .....	11
<b>9. References</b> .....	12

## **1. Abstract**

Protection of intellectual property in software development is required to stop people stealing critical code and bypassing product purchase. The Microsoft .NET framework is a fairly new technology, however, like Java, has worried many developers to the ease of decompiling its byte code. For example, the try-before-you-buy license is a popular licensing scheme utilised by both Java, and .NET programs, however it has been shown that they can be bypassed with little effort and knowledge (LaDue 1997). This paper investigates what intellectual property threats are prevalent in a try-before-you-buy licensed .NET Windows Application for marking student assignments and continue to discuss a form of tamper-proofing to make attacks of this type more difficult.

## **2. Introduction**

With advancements in technology, protection of intellectual property in the form of music, books and software is becoming increasingly harder to manage. Many companies are using copy protection mechanisms on their products to thwart people “stealing” their property and avoiding product purchase. However, this is not without flaws as people reverse engineer these products and remove any such mechanism, stealing critical secrets and bypassing product purchase.

With the development of the internet, a high emphasis was placed on software being able to run on multiple operating systems without having many different versions of the same product. With this came the birth of runtime compiled languages such as Sun Microsystems’ Java and Microsoft’s .NET Framework. However, to allow this platform interdependency, the data is stored in a strict manner with a large amount of metadata included with each application. This makes viewing and altering the code of these programs extremely straightforward and simple.

The internet also demanded applications to be downloadable online. To cope with this, some software companies have inherited a try-before-you-buy licensing scheme. This allows for a fully functional product to be evaluated for a short amount of time. After this time period the program is made dysfunctional, forcing the user to purchase the product for further use.

This privilege, however, opened up an opportunity for thieves to bypass this licensing limitation and use the software for free. When applications are natively compiled, this process is very time consuming and requires reasonable skills to understand the instructions made for the processor. However, with languages such as Java and .NET, bypassing these licensing limitations became extremely simple. Both languages provide tools to decompile their byte code into assembly

language and have also made recompiling simple, making the addition of circumvention code effortless.

To make understanding of the product code more difficult, obfuscation was born which attempted to secure the code by obscurity. Obfuscation attempts to replace unnecessary information embedded in the program code with less meaningful information. It may also change the way the program flows, or make obvious accesses to data less apparent. LaDue (1997) showed that obfuscation simply hid the fact that these languages' assemblies allowed easy disassembly regardless of whether obfuscated or not, by anyone who cared to inspect them.

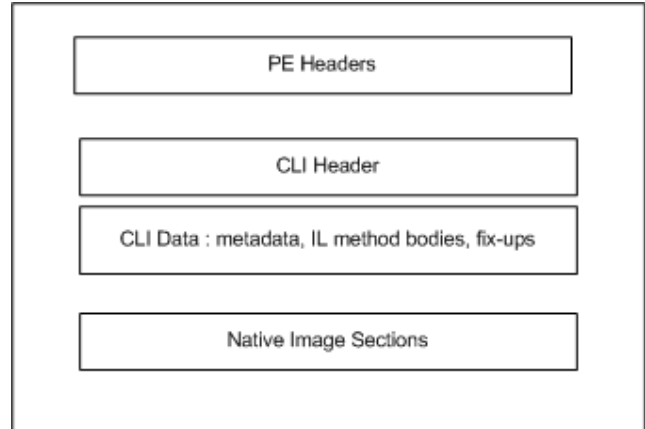
Plimmer and Mason (2004) have recently developed a marking application using the .NET framework called "PenMarked". This allows for digital annotations to be made on an electronic document with a pen input device. This application has sparked a lot of interest with academic and commercial institutions which has raised the worry of intellectual property theft.

This paper examines what information is stored in an application written for the .NET framework and performs an intellectual property threat analysis on PenMarked. A discussion is then provided as to how tamper-proofing can help guard against possible intellectual property theft.

### **3. A Brief Glance at Microsoft .NET Files**

The Microsoft .NET framework is still a relatively new technology, however has already gained popularity with developers due to language interdependency and ease of developing a functional windows application quickly. The .NET framework is based on a specification published by the ECMA TC39/TG3 on a standard called the Common Language Infrastructure (CLI). This specifies how a language should be written to enable cross platform and language interoperability with the .NET framework as an implementation of this standard. The CLI defines the layout of the .NET managed executables by using the Portable Executable (PE) File Format as used by Win32 executables.

When a .NET windows application is compiled, the result is an assembly file with an exe extension. This file is a strict extension to the Windows Portable Executable (PE) file format and contains both PE header information and platform independent byte code stored in little endian format. This file format "enables the operating system to recognize runtime images, accommodates code emitted as CIL [Common Intermediate Language] or native code, and accommodates runtime metadata as an integral part of the emitted code." A graphical representation of the .NET file format is shown in Figure 1. This file format is heavily described in the ECMA TC39/TG3 CLI specification under Chapter 24 of Partition II however is described in brief below.



**Figure 1**

*A graphical representation of a .NET Windows Assembly (ECMA TC39/TG3, 2002)*

The PE Headers of the .NET assembly are headers required by the Microsoft Windows Operating System. These headers identify the file to be a PE file, and contain information on how field values should be set. The CLI Header contains runtime specific information such as the address and sizes of the runtime data. The rest of the assembly contains the actual program data represented by metadata and an Intermediate Language (IL), specifically, Microsoft Intermediate Language (MSIL).

The metadata is “a structured way to represent all information that the CLI uses to locate and load classes, lay out instances in memory, resolve method invocations, translate CIL [Common Intermediate Language] to native code, enforce security, and set up runtime context boundaries.” This is where all new object types are declared and introduced into the Common Type System (CTS). It also contains information about the assembly manifest.

The manifest describes what items make up an assembly. It declares what files are present in the assembly and their relative offset, what object types are exported, and what assembly references are required to resolve unknown object types within an assembly.

The Intermediate Language inside each file in the assembly is stored as mapped byte codes. This means that all instructions can be directly translated to their assembly language equivalent. All primitive types inside a .NET assembly are stored in little endian format. String objects are stored in a Unicode encoding and are null terminated.

As .NET assembly files are structured strictly with rich metadata describing the runtime data, platform interdependency is achieved. However, combined with the manifest and program data, this means that disassembling a .NET assembly is extremely easy.

#### **4. Description of PenMarked**

The application chosen for threat analysis is a project developed by Plimmer and Mason (2004) titled “PenMarked”. This is an assignment marking system written in C#.NET based for Windows XP Tablet Edition. It utilises the pen provided with the Tablet Personal Computer and uses the Windows XP Tablet Application Programming Interface (API) for recognition and inking capabilities. The application allows the gap between traditional and electronic marking systems to be closed by allowing ink annotation onto electronic documents whilst retaining the benefits of electronic systems. It aims to create a highly effective working environment with the ability to process students faster and more efficiently than other methods, such as using databases or paper processing.

PenMarked currently spans across eleven assemblies, with over 200 classes and tens of thousands of lines of code. It is a very large project which contains some unique .NET components which may be a target for competitors. It is currently obfuscated using PreEmptives’ Dotfuscator Community Edition which simply renames variables and removes unused metadata.

The target audience for PenMarked is academic institutions who are interested in implementing an electronic assignment system, but wish to maintain traditional marking benefits such as being able to place annotations directly on a page. These academic institutions may be privately or publicly funded and range diversely in size.

PenMarked is still an early prototype without any licensing technology implemented. However, it is intended to have a form of try-before-you-buy option so the software can be downloadable from the internet and evaluated sufficiently.

#### **5. PenMarked Intellectual Property Threat Analysis**

An intellectual property threat in a software application is a situation where there may be a danger of users stealing code or the product for their own use. This will generally happen when the relative difficulty to steal the application or its code is less than the price to pay for the application. To prevent this type of threat, measures must be taken to make theft of this sort more difficult. It is much like protecting a valuable object by locking it inside a safe. The relative security applied to the safe must be greater than the cost to pay for the valuable object to avoid theft.

Two major intellectual property threats exist in PenMarked; bypassing licensing mechanisms and code extraction.

Due to the unique nature of PenMarked, it may be necessary to supply a try-before-you-buy option, downloadable from the internet. However, having this option opens up the opportunity for aversion

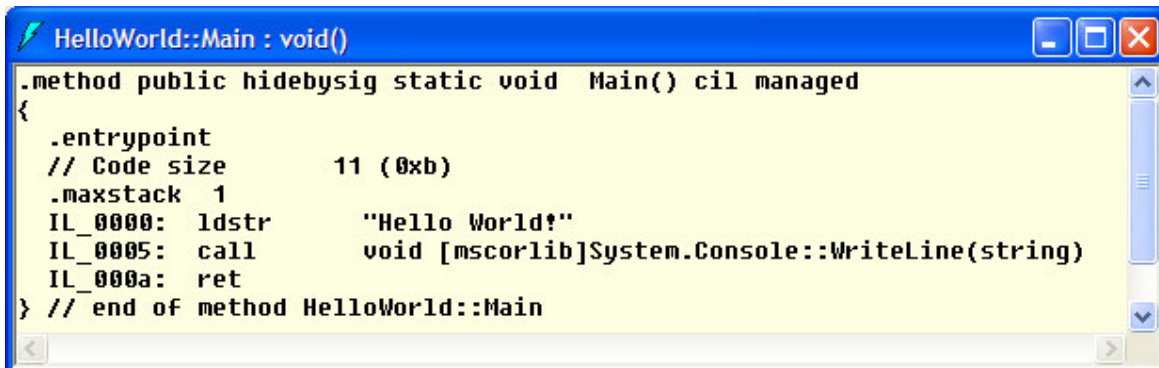
of product purchase. While one would expect academic institutions to keep within the law, smaller private institutions may not and thus attempt to avoid product purchase. It is not an option to provide a shareware copy of the software with limited functionality. To get an appropriate impression of PenMarked and to decide whether integration is possible, all features must be enabled.

LaDue (1997) demonstrated an extremely quick, straight forward method of bypassing virtually any licensing enforcement in Java programs. He argued that obfuscation did not make this diversion any more difficult, and that it simply “does little more than obfuscate an important truth about Java - the class file format allows free and easy disassembly of classes by anyone who cares to inspect and tamper with them”.

LaDue offered a few simple steps to circumvent three Java programs. The first step was finding the class file which contained the decision whether this programs license had expired or not. He did this by either investigating the installation script for a program, or by using a short UNIX shell script to search the class files for a give away string which the program displayed when demonstrating its limitations. Once the appropriate class file was identified, he used a simple tool provided with the Java Software Development Kit called “javap” which decompiles any java byte code into its appropriate assembly language representation. This code was then inspected for a licensing decision, which was often a simple branch instruction in the code. He then patched this class file using a quick tool that he had written. This tool replaced the branch with an instruction to skip this check, and then updated the constant tables to maintain the Java class file verifiability.

He concluded by suggesting that this type of licensing scheme is essentially futile. LaDue gave a name to this licensing method: the Maginot License. This was an appropriate name for these licenses as they are much like the French fortifications constructed between World Wars; easy to detect and to divert. A much better option, he suggests, is to provide interactive presentations over the internet or a form of shareware product with limited functionality. While shareware would be the ideal option, it is not practical for this application due to complete functionality necessary in order to properly evaluate the product. An interactive presentation would fail to present the added usage that this system could potentially provide, therefore the decision remains to build a try-before-you-buy system.

LaDue’s technique can map almost directly onto .NET assemblies. In fact, it is arguably easier and faster than using Java due to the rich .NET tools that are supplied with the development environment. There are two tools provided with the .NET framework to perform disassembly and reassembly: ildasm (Intermediate Language Disassembler) and ilasm (Intermediate Language Assembler). The disassembler has the ability to decompile an assembly which produces a windows resource file and a Microsoft IL assembly representation. The assembler has the ability to recompile



```
>HelloWorld::Main : void()
.method public hidebysig static void Main() cil managed
{
  .entrypoint
  // Code size          11 (0xb)
  .maxstack 1
  IL_0000: ldstr          "Hello World!"
  IL_0005: call               void [mscorlib]System.Console::WriteLine(string)
  IL_000a: ret
} // end of method HelloWorld::Main
```

**Figure 2**

*An example of the detailed assembly Ildasm.exe provides. Notice the clear call to the WriteLine method located in mscorlib.dll.*

an assembly given a resource file and an IL script. An example of the detailed information provided by the disassembler is shown via a simple 'Hello World' program in Figure 2.

While stealing the application is certainly a worry, another concern also presents itself; competitors stealing code. Currently, there are no other applications similar to PenMarked, yet it is inevitable that competitors will arise. Due to this relatively untouched area of education with computers and the odd behaviour of the ink API within the .NET framework, code extraction may be a real threat. Obfuscation already protects this to an extent, however due to the nature of the .NET framework code can still be extracted directly via the byte code. This requires some knowledge of the structure of IL code though with the rich information published by the ECMA, this is not hard to discover.

The technique used by LaDue to avert licensing mechanisms can be applied directly to the .NET framework and certainly poses a real danger due to the simplicity of the technique. However code extraction may require greater knowledge and more time to properly generate.

As performance of this application is not critical to system success, a small time and space overhead to combat these threats is acceptable. However, large time delays due to threat detection may disrupt workflow efficiency which is a saleable feature of PenMarked.

## **6. Tamper Proofing as a solution**

### **6.1 Overview**

To combat both intellectual property threats identified in PenMarked, tamper-proofing will be investigated. Tamper-proofing is the act of making the application crash or become unusable if any part of the program is modified.

There are three ways to detect tampering (Collberg and Thomborson 2002):

1. Examine the validity of the assembly using a one way digest algorithm such as MD5. If the original MD5 matches the current MD5 then the assembly is valid.
2. Check the validity of intermediate results produced throughout the assembly. This can be done by checking code segment digests or by checking algorithm results for correctness during the running of the program.
3. Prevent modification of the assembly by encrypting it. The decryption algorithm must be either protected in hardware, software obfuscated or both.

The MD5 matching algorithm, although potentially protecting licensing aversion, does not stop the theft of code from PenMarked. This method also fails as once decompiled, the single MD5 hash or other implemented digest can be located and altered to a new malicious value. Unfortunately, even storing the MD5 as an encrypted string via obfuscation does not help this attack. In the .NET and Java frameworks an obfuscated encrypted string can often be located via an instruction loading in a byte array followed by a jump to a decryption procedure. This method of detecting tampering is not acceptable for the threats outlined above.

Checking the validity of results throughout the program could certainly combat the problem of licensing aversion, but debatably stops code theft. Some previous work by Horne, Matheson et al. (2001), suggest making the testing critical to correct functioning. This could in effect stop some code theft such as a simple cut and paste attack. In this case, the code may be left unworkable due to a failed result check. An interesting extension to this method, suggested by Falcarin, Baldi et al. (2004) is by using Aspect Orientated Programming (AOP) to perform program checking on the fly. AOP is natively available in the .NET framework and thus makes it a tempting method to employ. The result set to check could be simply checking the message-digest of a segment of code throughout normal program execution.

By encrypting the assemblies, protection against both types of attacks specified by the threat analysis would be provided. Encryption would prevent people stealing code and also prevent any changes to the licensing mechanism. However, this would require a lot of time to adjust the

program to decrypt assemblies in memory and then dynamically load them. There is still a major flaw to this method however; the decryption algorithm. It is unrealistic for this software to employ a hardware mechanism to decrypt the assemblies at runtime. This would certainly render the try-before-you-buy licensing system unworkable and add an increased overhead to the cost of the product. However, using a software decryption method with the .NET framework would also be a problem. The decryption algorithm itself could not be decrypted, therefore open to attack. While multiple decryption algorithms could be used for each different assembly, identifying the decryption procedure would not be difficult due to assembly loading in .NET byte code being clearly visible. This would be shown clearly as a call to the "Assembly" class in the IL code due to the public visibility of this in a central .NET library. This would be similarly seen as the call to the WriteLine method as in Figure 2. To avoid this, the decryption library could be written in a natively compiled language such as C. The resulting DLL method could be called easily in .NET using an external method declaration, however this requires a decent amount of overhead to translate data types into the correct form. External method calls are also easily identifiable in .NET and would be able to point a hacker to the decryption class and entry point. This would help with disassembling the DLL library to discover the decryption algorithm and also ruin the capability of cross-platform interdependency if ever required.

With any method outlined above, it is necessary to implement antidebugging measures. This is to make it difficult to inspect the code while being executed by a debugger. This will avoid any sort of indications as to what is happening in the program and where. Obfuscation must also continue to be employed to make IL inspection more difficult.

## **6.2 The Verdict**

To actively protect against code theft and licensing aversion, the best method for PenMarked is to use a combination of both aspect orientated programming to check the validity of intermediate results, and encrypting some unique program assemblies. Although these methods will provide added protection to the project, one still has to consider the discovery and disablement attack.

Encrypting every assembly in PenMarked seems like overkill and will affect the runtime performance of the application. By not encrypting every assembly, this would save much overhead time and the trouble of altering the code to dynamically load all the libraries when needed. Avoiding the use of a natively compiled language will certainly leave an open wound in the application, however the overhead to add this feature would certainly not make this method worthwhile, so in order to minimise this problem special techniques must be employed. Currently the IL allows for global methods for compatibility with managed C++. Inserting an obfuscated global method directly into the IL code would essentially hide the decryption method from high-level decompilers, but still be available for those who cared to investigate the IL. For added security any decryption keys for the assemblies must be dynamically generated in memory rather than stored statically in the code.

Checking the validity of intermediate results throughout the program would actively thwart licensing attacks. A license check may be performed in multiple sections of the program, thus having this validity check would protect against such attacks as the one employed by LaDue. Using AOP to perform these checks utilises the .NET frameworks built in aspect features, but also exploits its speed. This would require placing simple checkpoints throughout the program which could dynamically check its runtime validity. To be predominantly effective, obfuscation must be used to hide the details of these checks.

Attacks at the runtime code are still possible, however, the time and money costs to properly thwart these attacks is greater than the price that PenMarked would be offered for. The disabling of any debugging processes is the limited amount of protection against runtime attacks that is accounted for.



### **6.2.1 Dynamic Self-Checking**

Horne, Matheson et al. (2001) suggested a method for dynamic self-checking of an application while executing, to verify that it had not been modified. They implement this technology by having two components; testers and correctors.

A tester is a component which checks a neighbouring section of code by computing a one way digest of its contents and comparing this with its own stored value. If these do not match then a response mechanism is started, either crashing the program, or exiting gracefully. Each tester is given a time interval and a code size to test. By adjusting these values gives a trade off between security and speed. For example, checking more often may provide a greater form of security however may slow down the system. Similarly with code sizes; a large code size computationally takes longer to derive and may affect the time complexity of the application. The code section that is tested always overlaps with other tester's code segments. This helps provide greater protection in the situation where one tester is successfully disabled. The testers are randomly assigned to each code segment in the program which provides another form of obscurity to thwart successful disablement.

The hash values of each segment of code are not stored with the tester for security reasons. Instead, a variable word called a corrector was implemented. Each tester has its own corrector, and each tester is itself tested by one other tester. A corrector is a single 32-bit unsigned integer which is placed in-between code blocks, thus not affecting the operation of the code.

Embedding these testers and correctors into the assembly was a three step process; Source-code processing, Object-code processing, then Installation-time processing. Source-code processing is simply entering testers, which are written in assembly code, into the executable. The testers are entered into the assembly source code as opposed to the object code to avoid using registers that may be used by the object. Object-code processing is a shuffle of the object blocks inside the executable, which essentially randomizes the distribution of the testers. This is when the correctors are inserted into the object code and the testers are assigned a corrector and a code interval to test. Installation-time processing consists of locating the watermark and re-computing correctors based on these. Finally installation was required applying patches containing these watermark and corrector values.

Horne, Matheson et al. (2001) provided an extremely technical method of tamper-proofing which is more technical than PenMarked requires. However it gives an underlying foundation of how a potential self-checking tamper-proofing could be implemented.

### **6.2.2 AOP Dynamic Self-Checking**

Aspect Orientated Programming is a natural extension to this dynamic checking mechanism and was theoretically discussed by Falcarin, Baldi et al. (2004). The solution they provide is essentially focused on a mobility problem where the code is downloaded and executed over a network. This is unneeded for PenMarked, however the principals discussed are highly relevant. Their work concentrated on an aspect orientated approach for Java which is relevant to the .NET framework due to many code and design similarities.

In their paper, they discussed to methods of AOP to provide a tamper-proofing mechanism: a static and a dynamic method.

The static method of AOP is where "hooks" are manually inserted into a program. These hooks can be used to call advice methods; that is, code which can perform validity checks on the current program executing. These methods have a range of different aspects available via the reflection class such as method and parameter information, class information and information about all global fields and their corresponding values. Among this, the method can also select subsets of code to perform validity checks on as the context of the aspect is separate from the object. If different aspect hooks are provided in the code then these can be also used to validate each other.

Dynamic hooks were also discussed by Falcarin, Baldi et al. (2004), however this required having a third party to insert hooks dynamically into the program aspect at runtime. The dynamic approach maintains the security aspect of static hooks, but also protects against discovery and disablement attacks. In the context of the article, it also relied on an external un-trusted entity to install and execute the application which poses a security risk for a brief amount of time. This particular solution requires an active network connection at all times for the dynamic retrieval of aspects in the program, however is impractical for an application such as PenMarked.

### **6.2.3 Assembly Encryption**

Encrypting some unique assemblies may provide a fair overhead to the application thus would have to be considered seriously. For the nature of PenMarked, an external native library is overkill, thus encryption would be performed using a global IL method. This would have to be manually written in .NET assembly language and also inserted manually to maintain utmost code obscurity. By doing so, this code could automatically load the assembly at runtime and return a reference to it for easy implementation. The disadvantage of this however, is that what is made easier for the developer, quite often makes it easier for an attacker also. While this method would protect the code on the surface, this method of encryption is essentially considered futile (LaDue, 1997). LaDue makes a suggestion that “applications protected by encryption is much like selling safes with lock combinations engraved on their bottoms”. This is because one can still search the code which uses this assembly and discover the “secret” call to this global method. However, a cost versus benefit analysis to implement another method does not justify the extra overhead. Obfuscation, in this case, is the best protection one could hope for against attacks at the decryption algorithm. And certainly, some sort of protection is better than none at all.

## **7. Discussion**

Horne, Matheson et al. (2001) discuss a tamper-proof system where a tester component continuously checks program segments for validity against a corrector. This type of method could easily be applied to a .NET program, however would not be as effective as the prototype they discussed. This is simply due to the openness of the .NET file format. Nevertheless, the dynamic self-checking system does not protect against a “detect and disable” approach. In defence, it is very difficult, if not impossible, for any method of tamper-proofing to be completely secure.

The level of security provided by the dynamic self-checking system is arguably too much for an application such as PenMarked. It may be more suited for an application of a more critical nature. However, it gives a basis of techniques which can be used to protect PenMarked in certain aspects, such as the location of correctors, and the checking of overlapping code segments.

By using an aspect orientated approach, the complexity of the dynamic self-checking approach can be removed and the .NET frameworks natural capabilities be further utilised. This approach combined with some ideas taken from the method suggested by Horne, Matheson et al. (2001) is the most viable option for PenMarked. Since AOP is a feature that is natively provided with the .NET framework via attributes, it is the simplest and quickest approach to implement. AOP would be the primary intermediary testing technique with the encrypted hash codes stored as global variables throughout the IL code. This can be easily achieved by editing the IL code manually and recompiling. For added security, the testers that were suggested by Horne, Matheson et al. (2001) could also be implemented at various positions through the code. This would of course be another trade off between performance and security however.

The static AOP approach suggested by Falcarin, Baldi et al. (2004), unfortunately is also open to “detect and disable” attacks. Attributes, while further hidden than normal code, can be removed with little effort. The only solution to this is to have multiple different hooks in the program and a high level of obfuscation to render the task of removing them too difficult.

The dynamic AOP approach suggested by Falcarin, Baldi et al. (2004) has a flaw to do with their trust model. They have assumed that the insertion of hooks at runtime will be completed by a fully

secure trusted platform which does not currently exist. They continue to elaborate that when the platforms become more secure, then this approach will be more viable. Whether this type of system ever evolves remains a question to be answered.

Using a static AOP to insert a self-checking mechanism is essentially the best method to go for PenMarked due to the simple mapping to the .NET framework. It may still be an insecure method, however it is difficult to find a tamper-proof system which is not.

The encryption technique has obvious flaws which are already well known (Ladue, 1997). By providing a software solution to decrypt the encrypted assemblies is an immediate security risk. It is similar to locking the front door of your home with the key hidden under the door mat. Anyone, with enough knowledge of where to look will eventually discover how to “unlock” your assembly. However, when there are limited options available, this is certainly better than no protection. If anything, it will protect against naïve hackers, but fail to stop the serious hacker.

If the assemblies are successfully decrypted, direct use of these will still not be possible. The .NET framework provides an option to have signed code permission sets for entities of an assembly. This will stop direct use of assembly if it was successfully unencrypted, however fail against code “cut and paste” attacks.

## **8. Conclusion**

Although tamper-proofing can help minimise the theft of intellectual property, it does not eliminate the problem completely. With enough time, effort and knowledge, it is still possible for a determined hacker to investigate the tamper-proofing mechanism and remove or alter it to use the application as they wish. However, the aim of tamper-proofing an application is to raise the difficulty of altering or stealing the program code so that it costs more in time and money to bypass these systems than to purchase or decompile the product. By adding a form of tamper-proofing to the application, this may protect against these attacks sufficiently.

This analysis has investigated three different forms of tamper-proofing an application to protect against intellectual property theft. While none of these methods are “fool-proof” they certainly protect the application further than its current state.

A form of Aspect Orientated Programming suggested by Falcarin, Baldi et al. (2004) will be used in combination with the tester and corrector components created by Horne, Matheson et al. (2001). The combination of these methods will not noticeably affect the computational time complexity of PenMarked and will more importantly provide a subtle method of constant protection against theft.

Encryption of more unique libraries will only protect the application on the surface, however will still be open to attack. This is more protection than is currently employed and mixed with obfuscation, may be sufficient to protect any would be theft of the intellectual property.

In conclusion, PenMarked will remain, in a sense, unprotected from intellectual property theft but its locks will have been upgraded, making it more difficult than before to bypass any protection mechanisms and steal this unique work.

## **9. References**

C. Collberg and C. Thomborson (2002). "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection." IEEE Transactions on Software Engineering **28**(8): 735-746.

ECMA TC39/TG3. (2002). "Common Language Infrastructure (CLI)". Available October 2004 <http://msdn.microsoft.com/net/ecma/>

P. Falcarin, M. Baldi, D. Mazzocchi (2004). "Software Tampering Detection using AOP and mobile code." AOSD 2004 Workshop, Lancaster United Kingdom.

B. Horne, L. Matheson, C. Sheehan, R. E. Tarjan (2001). "Dynamic Self-Checking Techniques for Improved Tamper Resistance." ACM Workshop on Security and Privacy in Digital Rights Management, Philadelphia Pennsylvania.

M. D. LaDue (1997). "The Maginot License: Failed Approaches to Licensing Java Software over the Internet." Available 1 October 2004 <http://www.geocities.com/securejavaapplets/maginot.html>

B. Plimmer and P. Mason (2004). "Designing an Environment for Annotating and Grading Student Assignments." OZCHI, Wollongong, Australia, ACM.