Watermarking Applications using Native Libraries

[#1]Akriti Kumari, [#2]Ashlin Mathew, [#3]Rutuja Jori
[#]Department of Computer Engineering,
Bharti Vidyapeeth's College of Engineering for Women,
Katraj, Pune - 411 043, India

ABSTRACT
In the present scenario, Intellectual Property Rights (IPR) for applications and software codes has become a primordial factor because of increasing level of software piracy. Software piracy is a direct threat to the revenue of software distributors and the country in general. A variety of preventive techniques have been developed for protection of the copyright of software codes or applications. But every technique developed till now is not strong enough to protect the software codes. In this paper, we propose a new watermarking technique for applications using Native Libraries and Java Virtual Machine. Firstly the libraries are written in C language called native libraries and then included in the Java code. The combining of Java and C languages is done using Java Native Interface (JNI). The Java Native Interface (JNI) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call, and to be called by, native applications (programs specific to a hardware and operating system platform) and libraries written in other languages such as C, C++ and assembly. When an application is written using native libraries, it becomes almost impossible to reverse engineer or decompile the application completely. After reverse-engineering an application written using Native Libraries to obtain the source code, the output will contain only the function prototype declarations.

General Terms
Application Security (Security by Design)

Keywords
Intellectual Property Rights (IPR), Java Virtual Machine, Java Native Interface, Native Libraries

1. INTRODUCTION
Software piracy refers to unauthorized use, copying or distribution of software. It is done by copying, downloading, sharing, selling, or installing multiple copies onto personal or work computers. Many people don't realize that when one buys software one is purchasing a license to use it and not the actual software. The license of the software tells us how many times the software can be installed. If one uses the software more than the license permits then it is piracy[2].

Software piracy is one of the direct threats to software industry which brings serious damages to the interests of software developers or providers and causes huge economic losses. One of the common types of software piracy is counterfeit software. Counterfeit software is a type of software piracy which is said to occur when fake software is produced and distributed in such a manner that it appears to be authentic. The Global Software Piracy Study 2011[4] states that the piracy rate globally, hovered at 42% while the market of pirated software has grown to $63 billion worldwide. As per the survey conducted by BSA, the piracy rate across the globe for past 5 years is as given in Table 1 and its comparison chart is as shown in Figure 1:

Table 1: Piracy Rate %

Region	Piracy Rate (%)				
	2007	*2008*	*2009*	*2010*	*2011*
Asia – Pacific	59%	61%	59%	60%	60%
C. & E. Europe	68%	66%	64%	64%	62%
Latin America	65%	65%	63%	64%	61%
M. East & Africa	60%	59%	59%	58%	58%
N. America	21%	21%	21%	21%	19%
W. Europe	33%	33%	34%	33%	32%
WORLD	**38%**	**41%**	**43%**	**42%**	**42%**

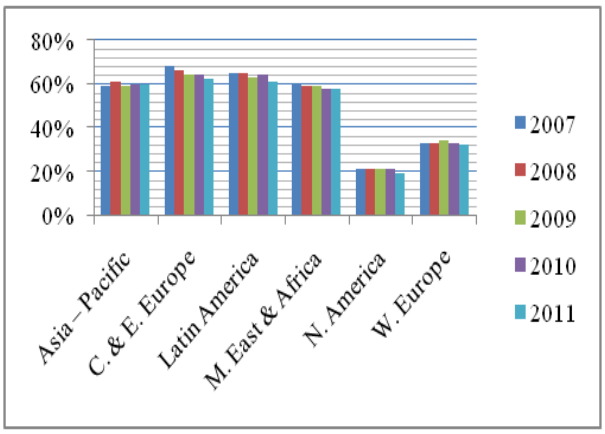

Figure 1: Piracy Rate (%)

The loss due to piracy world has been immense. Every year, software traders and companies have been losing millions of US dollars because of software piracy. The survey by Global Software Piracy Study 2011 states the loss in million US dollars[4] as given in Table 2 below and its comparison chart is given in Figure 2:

Table 2: Commercial Value of Unlicensed Software

Region	Commercial Value of unlicensed software ($M)				
	2007	*2008*	*2009*	*2010*	*2011*
Asia – Pacific	14,090	15,261	16,544	18,746	20,998
C. & E. Europe	6,351	7,003	4,673	5,506	6,133
Latin America	4,123	4,311	6,210	7,030	7,459
M. East & Africa	2,446	2,999	2,887	4,078	4,159
N. America	9,144	10,401	9,379	10,623	10,958
W. Europe	11,655	13,023	11,750	12,771	13,749
WORLD	**47,809**	**52,998**	**51,443**	**58,754**	**63,456**

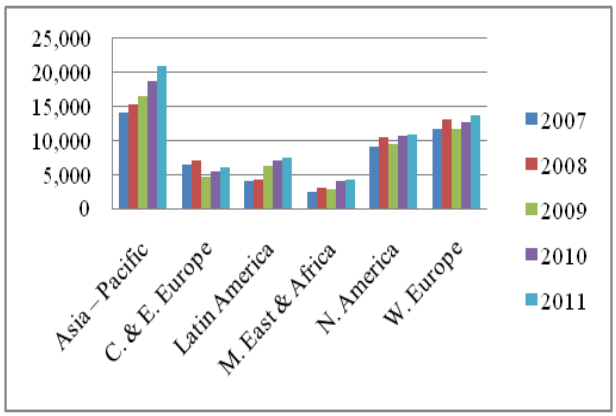

Figure 2: Commercial Value of unlicensed Software ($M)

2. WATERMARKING

A watermark is a marker embedded in an audio, video or an application's code with the aim to identify the owner. There are two different ways in which watermarking of applications or software watermarking is done: static watermarking and dynamic watermarking. Static watermarking embeds the watermark in the code or data section whereas dynamic watermarking stores the watermarks in a program's execution state, rather than in the program code itself.

2.1. Static Watermarking

Static watermarks are embedded in the code and/or data of a computer program,[1] a trivial example would be embedding a copyright notice in a string. Java class-files could contain software watermarks within their method bodies or their constant pool. The problem with storing a watermark as a string in a computer program is that unused variables could be easily removed with a simple dead-code analysis, and method or variable names are either lost during compilation or obfuscation. Other static watermarking techniques may involve transformations such as re-arranging or replacing instructions or basic blocks.

Static watermarking has various disadvantages. They are as mentioned below:

- Static watermarks are easy to construct and extract, but they are prone to attacks.
- Watermarks can easily be distorted by code obfuscators which may break up all the strings and scatter them around the program. General code obfuscation or optimization tools can easily change the static watermark, making them difficult to identify.
- Other attacks such as code motion and loop transformations can be used to distort the watermarks. Given the simplistic nature of static watermarks, these are relatively easy to distort and break.

2.2. Dynamic Watermarking

Dynamic software watermarks are inserted in the execution state of a program, rather than in the program code or data itself. It is shown that dynamic watermarking is more secret and robust, and has already become a main research direction present in software watermarking.[2] Prior techniques in dynamic watermarking hide the watermark in data structures that are built specially for this purpose during the execution of the program. The fact that the data structure is built specifically to house the watermark and it is independent of the application semantic makes the watermark susceptible to subtractive attacks. The behavior and semantic of the host program will not be affected by removing the data structures that hide the watermark[2].

Dynamic watermarks are of three types:

- Easter egg watermarks
- Data structure watermarks
- Execution trace watermarks

Easter egg watermarks display an image or a message after an unusual sequence of inputs is entered. Hence these are very trivial and easily be spotted[5].

Data structure watermarks are stored in the various data structures, the various instructions of the application; they can be extracted by checking the values of particular program's variables with a particular input sequence, simply by using a debugger tool[5].

Execution trace watermarks embed the watermark within the trace of application as it is executed with a special input sequence. It is used very rarely[5].

3. WATERMARKING USING JAVA

Recent developments in languages introduced coding of applications in Java language. Coding in Java is considered the most secure form of coding an application developed till now. Dynamic watermarking is done in the various libraries used in the application using Java. These files are then converted to .class files which are machine independent and hence can be executed on any machine. But this technique of Java based coding also has a flaw. If a software pirate acquires a Java Decompiler (JAD), every application developed in Java can decompiled easily.

3.1. JAD (JAva Decompiler)

A decompiler is a computer program that performs the reverse operation to that of a compiler; i.e. it translates a file containing information at a relatively low level of abstraction (usually designed to be computer readable rather than human readable) into a form having a higher level of abstraction (usually designed to be human readable). The decompiler does not reconstruct the original source code, and its output is far less intelligible to a human than original source code.

Jad (Java Decompiler) is such a decompiler for Java programming language. Jad provides a command-line user interface to extract source code from class files. A graphical user interface for Jad is JadClipse which is available as a plugin to the Eclipse IDE. Various versions of Jad are available on the internet, for e.g. *Mocha* is a Java decompiler, which allows programmers to translate a program's bytecode into source code.

The objective of Jad is to convert the .class files into Java source code. Most often a decompiler helps the programmers clarify poor documentation or provides a means for creating not yet written documentation. But on the other hand, decompilers are the prized components of any good software piracy kit.

Once the .class files are decompiled, the .java files are obtained. These files can then be edited to remove the watermarking and compiled again to give totally different .class files which will perform just like the initial .class files but will consist of no watermarking. As a result, the programmer would not be able to prove the ownership.

3.2. JNI (Java Native Interface)

To overcome the various shortcomings of Java based watermarking, the coding of various libraries to be used in an application can be done in JNI (Java Native Interface).

JNI is powerful framework for seamless integration between java and other programming languages (called native languages in JNI terminologies) used for improving performance and facilitating code reusability. JNI is extremely flexible, allowing Java methods to invoke native methods and vice versa, as well as allowing native functions to manipulate java objects. JNI renders native application with much of the functionality of Java allowing them to call java methods, access and modify java variables, manipulate java exceptions, ensure threat safety through Java thread synchronization mechanism and ultimately to directly invoke the Java Virtual Machine.[9]

JNI bridges the gap between Java and C. It does this by accessing shared libraries that can be written in C (or possibly C++).A *native method* is a Java method (either an instance method or a class method) whose implementation is written in another programming language such as C.

3.2.1. JNI Data types:

It is essential that the data types that are passed between Java and native code have the same properties. The easiest and safest way to use the table below when mapping between Java and native code.

Table 3: Mapping Between Java and Native Data Type

Java Type	Native Type	Description
boolean	Jboolean	8 bits, unsigned
byte	Jbyte	8 bits, signed
char	Jchar	16 bits, unsigned
double	Jdouble	64 bits
float	Jfloat	32 bits
int	Jint	32 bits, signed
long	Jlong	64 bits, signed
short	Jshort	16 bits, signed
void	Void	N/A

3.2.2. JNI Type Signatures

A *signature* is a list that specifies a class constructor, an instance method, or a static method, and thus distinguishes it from other constructors, instance methods, or static methods. A *simple signature* is a single element list containing the name of the method or constructor. In most cases a simple signature is only needed as the Java method resolver is able to disambiguate overloaded Java methods based on the types of Java object arguments. The *full signature* is used to distinguish between two or more methods or constructors that have the same number of arguments. In native code, Java type signatures are encoded using the mappings shown below [6] [7].

Table 4: Java Type Signature Encoding

Java Type	Signature
boolean	Z
byte	B
char	C
double	D
float	F
int	I
long	J
void	V
object	L*fully-qualified-class*;
type[ ]	[*type*
method signature	*(arg-types) ret-type*

4. INTEGRATION OF NATIVE CODE USING JNI

Let us understand the integration of native code into Java programs using the simple example of 'Hello World!' application [7].

1. Create the Java files. Create the two Java files as shown below.

HelloWorld.java

– declares a native method

```
class HelloWorld
{
        /* declare native method */
        public native void displayMessage();
        static
        {
          /*Load the Library*/
           System.loadLibrary("HelloWorldImp")
       ;
        }
}
```

– create a HelloWorld object and call the native method

```
class Main
{
        public static void main(String[] args)
```

```
        {
            HelloWorld hello = new HelloWorld();
            hello.displayMessage();
        }
    }
```

2. Compile the Java files**.**

```
javac HelloWorld.java
```

3. Create the header file. Uses the .class file created previously to create HelloWorld.h. (Note the -jni argument to javah.)

```
javah -jni HelloWorld
```

This produces the file HelloWorld.h. It contains the C declarations for the methods that were declared native in HelloWorld.java. If we open the HelloWorld.h file we will see the following code[8]:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloWorld
 * Method:    displayMessage
 * Signature: ()
V
 */
JNIEXPORT void JNICALL Java_HelloWorld_displayMessage
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

• javah - **j**avah is a useful tool that creates a C-style header file from a given class. The resulting header file describes the class file in C terms. Although it is possible to manually create the header file, this is almost always a bad idea. javah knows exactly how Java types and objects map into C types.

• jni.h - jni.h is a C/C++ header file that is included with the JDK. This file defines all of the necessary data types. It contains mostly #defines and typedefs that hide the complexity of mapping Java types to native types. It also #includes many other platform-specific header files. Since javah automatically includes this header file, you as the programmer usually do not have to explicitly include this file.

4. Create the C file.

The function that you write must have the same function signature as the one you generated with javah into the HelloWorld.h file.

• HelloWorld.c

```
#include <stdio.h>
#include "HelloWorld.h"    // this header file was generated by javah
JNIEXPORT void JNICALL Java_HelloWorld_displayMessage(JNIEnv *env, jobject obj)
{
        printf("Hello World!\n");
}
```

5. Build the shared library from native code.

This is where some of the differences show up. Make sure that the search paths are correct and that the file extensions are what the compiler expects. Also, the convention in UNIX is to prepend lib onto the names of library files and to append .so onto the names of shared library files.

- This works on sirius.cs.pdx.edu, note .C extension

```
g++ -G -I/pkgs/jdk1.1.1/include -I/pkgs/jdk1.1.1/include/solaris HelloWorld.C -o libHelloWorldImp.so
```

- This should work under NT. Replace the include paths to match your environment and be sure to include Sun's JDK not Microsoft's!

```
cl -Im:\jdk1.1.5\include -Im:\jdk1.1.5\include\win32 -LD HelloWorld.c -FeHelloWorldImp.dll
```

6. Execute the program.

On UNIX, you may have to move the shared library into a directory that's in your search path, or add the current directory to your path. Under Windows NT, the current directory is searched by default.

```
java Main

Hello World!  ← This is displayed on the screen
```

5. DECOMPILATION OF JNI CODED APPLICATION

Java bytecode is very easy to decompile - just Google for "download java decompiler" and one will get his/her source code back in minutes. In contrast, native code is about as hard to reverse engineer as if you have coded the original program in C\C++. Also, there is no performance loss. If real owner is concerned about protecting its intellectual property, native methods is the safest way to proceed.

After compiling a code in JNI, on decompiling the Java code the output will contain the Java code which only has function calls but that code cannot reveal the library code written in C i.e. the body of the function in the library. If dll file is decompiled separately it will have signatures that belong to JNI. So unless a person is capable enough of rewriting those libraries (in that case there is no need for using our libraries),

it is not possible to reuse the code by removing the watermark.

Thus the code of the software programmer is safe. Whoever wants to use the application under some other name, will have to write the whole library; in that case there is no use of decompiling our code.

6. THREAT ANALYSIS

6.1. Additive Attacks

An additive attack means inserting another watermark into an already authenticated application, which may or may not over-write the existing watermark. With current watermarking algorithms, an additive attack sometimes removes the original watermark if a watermark of the same type is embedded but not necessarily if a different type of watermark is embedded. If the original watermark is not over-written by the new watermark the attacker could claim ownership of the software if both watermarks are recognizable, resulting in a dispute over ownership. The only possible solution is to register the watermarks with third party and timestamp them[2]. With this method, the real owner can claim that his/her watermark is original.

6.2. Subtractive Attacks

In this type of attack, attacker tries to remove watermark from the software program. However by doing that he might damage parts of program or some of its functionalities. Subtractive attacks cannot be invoked against the proposed watermarking technique because the watermark is not stored as any data structure, occupying no physical space; rather, it is encoded in the runtime code of the hash function[2]. The attacker might try to invoke attack by deleting the whole hash function. In order to accomplish this it is necessary to:

- Replace all calls to hash function by the constant. Value returned by that function.
- Modify/remove the hash function

There are so many function calls in program and the attacker cannot be sure that he has located the right function. In our implementation of hash function, different value is returned according different input, and the algorithm we choose is irreversible.

6.3. Distortion Attacks

In these types of attacks, the attacker might be able to damage or distort the watermark in some manner that the real owner cannot prove his ownership of the code. Usually, it is assumed that the attacker does not know the location of the watermark which means they would have to apply all distortive attacks uniformly across a program; this is likely to have an adverse affect on performance [1]. In the proposed methods if the attacker will change of one's defined random function then we can identify changes by another random function.

6.4. Collusive Attacks

Collusive attacks involve the comparative analysis of 2 or more copies of a *fingerprinted* program. In a simple case the only difference between the two watermarks would be the fingerprint thus revealing the location of the fingerprint in all the programs. Every program must be obfuscated differently before distribution in order to avoid this kind of attack. After obfuscation there will be many differences between the programs and the watermark will be harder to find. However, this may cause a problem for debugging customers' programs; for example, bug reports sent in by customers may be specific to their copy of the software. Collberg and Thomborson suggest that it will be necessary to store a copy of the keys used to fingerprint and obfuscation every copy of a sold program in order to recreate and exact copy of the customers program for debugging purposes [3].

7. CONCLUSION

The proposed watermarking technique of coding applications which uses Native methods in Java allows the developer to protect his software for pirates who misuse the source code. Through this technique any software application can be made secure and also improve its functionality. Software codes stored the library with is created using JNI is not available when an application is downloaded. Accompanying the proposed technique with a robust dynamic watermarking technique will result in an application which cannot be decoded at all and hence is completely protected against piracy.

8. ACKNOWLEDGEMENTS

This work would not have seen the light of the day without the whole-hearted support of our guide Prof. Sonali Kadam. We also extend our sincere and wholehearted thanks to the Head of Department Prof. Dr. S. R. Patil and all the members of the Department of Computer Engineering, Bharati Vidyapeeth's College of Engineering for Women, without their support and inspiration this would not have been possible.

9. REFERENCES

[1] B.K.Sharma, R.P.Agarwal and Raghuraj Singh, "Copyright Protection of Online Application using Dual Watermarking", International Journal of Computer Applications, Volume - 18, Number 4, Article 5, March 2011

[2] Xuesong Zhang, Fengling He, Wanli Zuo, "Hash Function Based Software Watermarking" IEEE International conference on ASEA 2008, 13-15 Dec. 2008, PP 95 - 98

[3] Yawei Zhang, Lei Jin, Xiaojun Ye, "Software Piracy Prevention: Splitting on Client", IEEE International Conference on Security Technology, December 13-15, 2008, PP 62-65

[4] Business Software Alliance. http://www.bsa.org

[5] Christian S. Collberg, Clark Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation Tools for Software Protection", IEEE Transactions On Software Engineering, Vol. 28, No. 8, August 2002, PP 735-746

[6] J. Hamilton, "A survey of static software watermarking", IEEE World Congress on Internet Security 2011, PP 100 - 107

[7] Mathew Mead , "Programming in C/C++ with Java Native Interface", http://home.pacifier.com/~mmead/

[8] Christopher Batty, "Using The Java Native Interface" Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada, www.cs.umanitoba.ca/~eclipse/8-JNI.pdf

[9] Evgeniy Gabrilovich, Lev Finkelstein, "JNI – C++ integration made easy", C/C++ Users Journal, Volume 19 Issue 1, Jan. 2001, PP 10-21