UWStego: A General Architecture for Software Watermarking

C. Collberg[*] S. Jha[†] D. Tomko[†] H. Wang[†]

August 31, 2001

Abstract

Software piracy is a cause of substantial losses for software vendors. For example, software and technology piracy is suspected to cause approximately $16 billion each year. Given the magnitude of losses due to software piracy, companies need ways to prosecute software pirates. An essential step in proving the guilt of a suspected software pirate is to trace the source of a program, i.e., that a specific program originates from a certain company. *Software watermarking* is a technique that can be used for identifying the source of the program. In this paper, we survey various software watermarking techniques. We present several metrics to gauge the efficacy of various software watermarking schemes. Finally, we present the design and implementation of a general architecture, `UWStego`, for watermarking JAVA programs.

1 Introduction

Suppose that the Acme Software Company develops and markets a program and then sells the distribution media and software licenses to customers. At some point along the distribution chain, a software pirate obtains a copy of the media. The pirate proceeds to make illegal copies and distributes them through his or her own channels, possibly through the internet, or by "burning" the software onto writable compact discs. The illegal copies may clearly be pirated if, for example, the software is offered for internet download, or the copies may be packaged as an attempt to counterfeit the original, and to thereby deceive the customer as well as law enforcement authorities.

The Acme Software Company, having made a large investment in the design and development of the program, is injured by the activity of software pirates. The program's legitimate sales are affected, and thus the ability of the company to develop new software is diminished. Not only does

[*]Department of Computer Science, University of Arizona, Tucson, AZ 85721.
[†]Computer Sciences Department, University of Wisconsin, Madison, WI 53706.

this directly hurt the company and it's investors, but the ripple effect through the economy can be tremendous. For example, Los Angeles County Sheriffs discovered $8.5 million in counterfeit software in one search alone. Globally, software piracy and other technology piracy is suspected to be a $16 billion business each year [Gla00].

Given the magnitude of the estimated losses, software producers need methods to discourage the practice of piracy, and to aid in prosecuting criminals when it takes place. The first step in bringing a suspected software pirate to justice is to prove the identity of any copies suspected to be stolen. *Watermarking* is a mechanism which can be used for program identification. If a software producer embeds a watermark in a product before it is distributed, then at some later date this watermark can be exposed. This demonstrates that the origin of the program is the producer. This evidence can then be used in court as direct proof of the ownership of the software copyright. Alternatively, it can be used prior to a court case to obtain depositions and subpoenas in order to gather further evidence.

If a product is watermarked, the pirate will try to find techniques for removing, distorting, or destroying the watermark to prevent subsequent identification. Of course, the software producer wants to structure the watermark so that it is either as difficult as possible to remove, or so that tampering with the watermark destroys the usability of the program.

At the highest level, there are two watermarking paradigms, the most basic of which is the embedding of a single watermark in the final production copy of the product, which insures that all copies sold contain the same mark. This only protects against the threat of another party claiming that the origin of a copy is not the actual producer. In this case, the existence of the producer's watermark can be demonstrated, and thus prove ownership of the article of intellectual property.

The second paradigm is slightly stronger, because when exposed the watermark provides more information than simply the origin of the software. In *serial watermarking*, known in prior literature as *fingerprinting*, a different watermark is embedded in each copy sold, and the producer notes at the time of sale the message text in that particular copy and the customer to which it is sold. If an illegal copy is suspected and a watermark can be detected in it, then the owner of the software can be confirmed and as well as the source of the illegal copy. This may allow for more evidence to be collected, thus strengthening a court case against the user or distributor of the illegal copy.

The methodology for software watermarking can also be divided into two major types, *static* and *dynamic*. Systems that encode the watermark data directly in the program executable are static systems. The watermark may be stored in any part of the executable, so long as the semantics of the program are preserved. For Java programs, the constant pool, the method or field tables, or any attribute may be used to encode the watermark. To detect the watermark, the program executable is statically analyzed by a *decode function*, searching for the watermark data. Instead of encoding the watermark data directly in the text of the program executable, some systems add code to the program which constructs the watermark in the runtime state of the program. Such systems were first proposed by Collberg and Thomborson [CT99a] and are called *dynamic systems*. To detect the watermark, instead of analyzing the program directly, some other artifact of the program is searched such as a profile of the run time state of the program. A schematic diagram for software watermarking is shown in Figure 1.

In this paper we explore dynamic watermarking systems. This paper makes three major contributions to the field of dynamic software watermarking.

- First, we survey and classify existing software watermarking techniques. Weaknesses of these techniques are clearly discussed. Moreover, threat models for software watermarking systems are also discussed in great detail.

- Software metrics is a vast area [FP98], but the focus of a significant amount of research on software metrics is geared towards process. We believe that metrics for software watermarking are fundamentally different from metrics for other areas of software engineering because of the presence of malicious attacker. We present various metrics for gauging the efficacy of various software watermarking schemes. The subject of metrics has received scant attention in the software watermarking literature, but is crucial to a systematic development of the field.

- Finally, we present a general architecture, `UWStego`, for watermarking JAVA programs. Our architecture enables a software engineer to easily implement new dynamic software watermarking techniques for JAVA programs. Modules in the `UWStego` architecture are designed using well established principles of information hiding [Par72]. We also discuss how this general architecture can be used to boost the stealth and resilience of software watermarking systems.

2 Definitions and Metrics

A *steganographic system* is a method which hides a piece of extra information inside another [Way96]. The goal of such a system is to disguise the existence of extra data, called the *message text*, within the *cover text*. The message text can be retrieved or *exposed* at some point in the future in order to communicate the desired information.

The term *watermarking system* refers to a steganographic system when it is used to assist in the protection of intellectual property by using the message text as a hidden, identifying mark. The intent is not to communicate the message text to another party, but to use the existence of the message text, when necessary, as a proof of origin. To this end, the message text is usually short, perhaps something like `‘‘Copyright 2001 ABC Corp’’` represented as an ASCII encoded string. Such

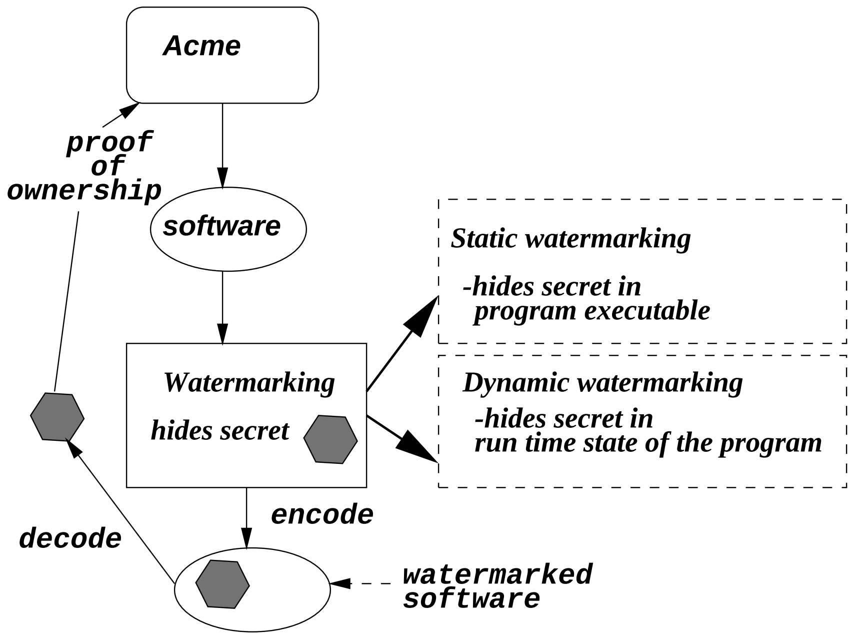

Figure 1: Schematic diagram of software watermarking.

a short message text helps to keep it's presence imperceptible, but is still enough to be statistically unlikely to appear by coincidence.

Recently, most research on steganography has focused on embedding message text into various forms of digital objects. Since any digital object is simply a string of bits, a steganographic system for such objects can be treated in a mathematical fashion. A *steganographic system* is defined by a five-tuple, $S = (\mathcal{O}, \mathcal{W}, \mathcal{O}_\mathcal{W}, \mathcal{E}, \mathcal{D})$,

1. $\mathcal{O}$ is the set of objects which may be used as *cover texts*.

2. $\mathcal{W}$ is the set of valid message texts, or *watermarks*.

3. $\mathcal{O}_\mathcal{W}$ is the set of *cover texts* with an embedded message.

4. $\mathcal{E} : \mathcal{O} \times \mathcal{W} \rightarrow \mathcal{O}_\mathcal{W}$ is the *encoding function* that embeds a message text in a cover text.

5. $\mathcal{D} : \mathcal{O} \rightarrow \mathcal{P}(\mathcal{W})$ [1] is the *exposition function* which attempts to detect and expose any message texts contained within $\mathcal{O}_\mathcal{W}$. Notice that the exposition function returns a set of "potential" watermarks.

[1] $\mathcal{P}(\mathcal{W})$ denotes the power set of $\mathcal{W}$.

For our discussion of software watermarking systems, we will consider $\mathcal{O}$ to be the set of programs, $\mathbb{P}$, or some subset of $\mathbb{P}$, such as the set of Java programs. For any program, $p \in \mathbb{P}$, and a message text, $w \in \mathcal{W}$, the watermarked program $\mathcal{E}(p, w)$ is denoted by p_w.

A watermark, $w \in \mathcal{W}$, is *exposed* by $\mathcal{D}$, and therefore $\mathcal{D}$ is said to be successful, if $w \in \mathcal{D}(p_w)$. Assume that we are given a program $p \in \mathbb{P}$ and a watermark $w \in \mathcal{W}$. If p is not the result of watermarking a program with w and $w \in \mathcal{D}(p)$, then we have a *false alarm* event (denoted by $E_{\text{falseAlarm}}$. The expression given below formalizes the false alarm event.

$$w \in \mathcal{D}(p) \land p \notin \mathcal{E}(\mathbb{P}, w)$$

In the expression given above $\mathcal{E}(\mathbb{P}, w)$ is the set of all programs obtained by watermarking with w, or formally

$$\mathcal{E}(\mathbb{P}, w) \quad = \quad \{p_w \mid p \in \mathbb{P}\}$$

Now if we have a probability distribution over the set of programs $\mathbb{P}$ and watermarks $\mathcal{W}$, then we can compute the *false alarm rate* (denoted as α) as the probability of the false alarm event $E_{\text{falseAlarm}}$. Naturally, a watermarking system must be designed so that α is acceptably small. Likewise, without the presence of an attacker, the exposition function acting on a program that contains a watermark embedded by the corresponding encoding function $\mathcal{E}$ should be successful with a very high probability. Assume that we are given a program p and watermark w. The expression given below characterizes the *hit* event E_{hit}, i.e., an embedded watermark is successfully exposed

$$w \in \mathcal{D}(\mathcal{E}(p, w))).$$

Now again if we have a probability distribution over the set of programs and watermarks, we can compute the *hit rate* (denoted as β) as the probability of the hit event E_{hit}. The aim of a successful watermarking system is to keep the hit rate β high and the false alarm rate α low.

An attack on a watermarking system is any function, $A : \mathbb{P} \rightarrow \mathbb{P}$. We will denote the set of possible attacks as $\mathbb{A}$. A *threat model*, $\mathcal{M}$, for the purpose of watermarking research is a set of attacks, $\mathcal{M} \subset \mathbb{A}$. We will discuss various strategies and specific attacks in Section 5.

For this paper, we will use $S(p, I, t)$ to denote the runtime state of a program, p, (which may or may not be watermarked) on input I, at program point t. For current dynamic watermarking implementations on Java programs, typically t is the end of the program, which we will denote as "`end`."

A person whose intent is to remove, distort, or destroy a watermark is said to be an *attacker*. An attacker transforms a watermarked program so that the exposition function cannot recover the watermark, i.e., given a watermarked program $\mathcal{E}(p, w)$ an attacker chooses an attack $A \in \mathbb{A}$ such that

$$w \notin \mathcal{D}(A(\mathcal{E}(p, w)))$$

Once the attacker has destroyed the watermark, he/she can insert a new watermark w' using the encoding function $\mathcal{E}$[2]., i.e., the attacker uses the following operation:

$$\mathcal{E}(A(\mathcal{E}(p, w)), w')$$

2.1 Metrics

Before describing current software watermarking systems and attack strategies, we need metrics for evaluating these systems. Collberg and Thomborson give definitions of stealth, resilience, and data rate applicable to their dynamic graph technique [CT99a]. We generalize these definitions and also add a definition for the feasibility of a software watermarking system.

The *feasibility* of a watermarking system measures the impact of embedding a watermark in a program. Specifically, the watermarked program must still fulfill the functional requirements of the original, but there may be some performance impact. For any program, we need some function, $V : \mathbb{P} \rightarrow \Re$, to measure it's performance. A lower number indicates better performance, possibly in terms of execution time or memory requirements. The producer must choose some acceptable impact, δ, and choose a watermarking system such that for all p and w, $\frac{V(p_w)}{V(p)} \leq \delta$, i.e., watermarking system does not slow down a program by more than a factor of δ.

The *data rate* measures the number of message text bits which are encoded per bit added to the target program, p. The size of the target program may be increased in two ways: the size of the executable in a storage medium, and the memory used by the program at run time. These correspond to the static and dynamic data rates, respectively. The *static data rate*, $R_s(\cdot, \cdot)$ is measured by comparing the size of the original and watermarked executables with the size of the watermark. For a program $p \in \mathbb{P}$, and a message text $w \in \mathcal{W}$ the data rate $R_s(p, w)$ is given by

$$R_s(p, w) \quad = \quad \frac{|p_w| - |p|}{|w|}$$

As mention before, if we have a distribution over $\mathbb{P}$ and $\mathcal{W}$ we can define the *average static data rate* as the expected value of $R_s(p, w)$.

Additionally, the presence of the watermark may impact the size of the runtime state of the program. This impact is measured by the *dynamic data rate*, which is the ratio of the number of bits added to the runtime state to the number of bits in the watermark. Given a program p and

[2]Notice that we assume that the algorithm for the encoding function $\mathcal{E}$ is public.

message text w the dynamic data rate $R_d(p, w)$ is defined as:

$$R_d(p, w) \quad = \quad \max_{I \in \text{In}(p), t \in \text{PP}(p)} \frac{|S(p_w, I, t)| - |S(p, I, t)|}{|w|}$$

In the equation given above In(p) and PP denotes the set of inputs and program points corresponding to p. Notice that we compute the maximum over all inputs and program points.

Collberg and Thomborson give a simple statistical measure of stealth which looks for instructions and sequences of instructions that are unlikely to occur in the normal program, and therefore may be part of a watermark [CT99a]. This is interesting for systems that add to or change the program's code, but is not applicable to other types of schemes.

We consider *stealth* to be the difficulty of gathering information about the watermark. Before making some modification to the program in an attempt to remove or otherwise destroy a watermark, the attacker can be aided by some information about the location or nature of the watermark. Once an attacker has the distribution media for a program, there is no limit on what that he/she can do to learn about the program and the watermark. Even reverse engineering via a disassembler or decompiler is applicable in attempts to learn about the watermark. Because of the wide variety of non-mathematical information gathering attacks on watermarks, we give no equations for calculating a value for stealth, but it is important for watermarking system designers to understand the power and flexibility of these information gathering attacks.

We define *resilience* of a watermarking system to be the ability of the system to successfully expose watermarks after an attack attempt has been made. We will only consider algorithmic attacks which may be automated when measuring the resilience, because if an attacker can learn enough about a watermark, then it can be cut out exactly by hand. If the right information can be gathered, no system can withstand this type of attack. This definition of resilience is generalized from that given by Collberg [CT99a], who only considered attacks which added extra information to either the program executable or to the runtime state of the program.

First, we define resilience with respect to one attack. Subsequently, we extend the definition to multiple attacks. A static watermarking system is resilient to an attack $A \in \mathbb{A}$ if the probability of the following event is low [3]:

$$w \ \notin \ \mathcal{D}(A(E(p, w))) \ .$$

$A(E(p, w))$ denotes the program obtained after applying the attack A to the watermarked program $E(p, w)$. The condition given above states that the valid watermark w cannot be recovered by the exposition function D. Therefore, the attack A was successful in removing the watermark.

The corresponding definition for dynamic watermarking systems requires one additional parameter, t, which is the program point specifying the time at which to observe the program state.

[3]The precise definition of low probability depends on the context. Usually low probability is a probability below a certain threshold.

Similarly, a dynamic watermarking system is resilient to an attack $A \in \mathbb{A}$ if the probability of the following event is low:

$$w \ \notin \ \mathcal{D}(S(A(\mathcal{E}(p, w)), I, t)).$$

A watermarking system is said to be *secure* with respect to a threat model, $\mathcal{M}$, if the system is resilient to each attack, $\mathcal{A} \in \mathcal{M}$, and furthermore is resilient to any composition of the attacks in $\mathcal{M}$. Define the *closure* of $\mathcal{M}$ or $\mathcal{C}(\mathcal{M})$ to be:

$$\mathcal{C}(\mathcal{M}) \ = \ \{g_1 \circ g_2 \circ ... \circ g_n \mid g_1, ..., g_n \in \mathcal{M}\}$$

A static system is secure with respect to a threat model $\mathcal{M}$ if and only if it is resilient with respect to each attack $A \in \mathcal{C}(\mathcal{M})$. This definition logically extends to dynamic systems in the same manner as for resilience with respect to a single attack. If we do not allow the attacker to compose an arbitrary number of threats, then we can bound the length of the chain of compositions to be less that a specific number k, e.g., we define a *bounded closure* $\mathcal{C}(\mathcal{M})_k$ as

$$\{g_1 \circ g_2 \circ ... \circ g_n \mid g_1, ..., g_n \in \mathcal{M} \text{ and } n \leq k\} \ .$$

A watermarking system is resilient with respect to the bounded closure $\mathcal{C}(\mathcal{M})_k$ if it is resilient with respect to each attack $A \in \mathcal{C}(\mathcal{M})_k$.

3 A Survey of Current Watermarking Systems

3.1 Static Watermarks

A United States patent issued to Holmes [Hol94] in 1994 describes a simple static watermarking method. In this system, the master copy of the program contains a segment of data which is not used by the program. The location and size of this unused segment is determined when the program is linked. When a copy of this program is made for authorized distribution, this segment is overwritten with the watermark information, such as the date, time, and destination of the copy. Holmes proposes this method as being suitable for Internet distribution of a program, since a serial watermark may be easily embedded by a server providing customers with copies as they are purchased.

A nearly identical system has been proposed by Samson [Sam94], which also utilizes an unused data section. However, instead of overwriting this section with arbitrary watermark data, Samson's watermarks consist of three integers that exhibit an unusual property. A patent was issued for this system in 1994, which uses one function to generate the watermark to be embedded in the program. This function is keyed on a public program identification number that the producer assigns to each product, and also a private key that the producer holds in secret. Program numbers and corresponding customers are recorded by the producer for later use as evidence in case an unauthorized copy is discovered. An additional method is added to the program which checks the property of the watermark at runtime as a tamper-proofing measure. Since this property is unusual, it is statistically unlikely to occur in three randomly chosen numbers. If the property holds, the program is allowed to execute, otherwise the program terminates.

Monden *et al.* [MIM$^+$00] describe a method for watermarking individual Java classes that also embeds the watermark in unused data, but instead of adding arbitrary static data, they add extra code to the program. Their idea has been implemented in a tool called *JMark* [Mon00]. Their system requires that the programmer include an extra method in each class to be watermarked and place a call to that method somewhere else in the code. This call should be conditioned on a predicate which always evaluates to false, but static analysis techniques should not reveal this information. Such predicates are called *opaque predicates* [CTL98]. Predicating the call on an opaque predicate is an attempt to prevent dead code analysis from locating and removing the method containing the watermark. The watermark is then embedded in the extra method by changing operands and opcodes such that the rules of well-formed classfiles are not violated. Since the extra method is never be called, the semantics of the class is unchanged. To encode the watermark, JMark treats the message text as a stream of bits and encodes these bits in the extra method as it find opportunities to do so. If JMark finds a bytecode that has an immediate operand, it replaces the bits of the immediate field with an equal number of message bits. For example, the `iinc` instruction takes two eight bit parameters, first, the index of an integer local variable, and second, a signed integer value. The latter parameter may be replaced without disrupting the validity of the classfile.

Additionally, there are also opcodes that may be replaced. JMark has a table of equivalence classes of Java bytecodes. The bytecodes in each equivalence class all share the same syntax, require identical operand stack pre-conditions, and produce identical operand stack post-conditions. This property guarantees that the validity of the classfile is preserved if any opcode in the set is replaced with any other. For example, the `iadd`, `isub`, `imul`, `idiv`, `irem`, `ishl`, `ishr`, `iushr`, `iand`, `ior`, and `ixor` instructions all have no arguments, take two integer operands from the stack, and push an integer result. The sets are trimmed to a size that is a power of two. For each opcode in an equivalence class, JMark assigns a binary number. An example of such an assignment is shown in Table 1. From these tables, opcodes may be changed to encode a number of message bits equal to the binary logarithm of the size of the class.

Moskowitz [MC98] proposes a method of static software watermarking that draws from media watermarking. In his system, the watermark is embedded in an image, which is then placed in the static data segment of the program. In the case of this patent, the watermark data is not a simple string, such as, `"Copyright 2001 Acme Corp."`, but instead is an essential piece of the code for the program. At runtime, the code is extracted from the image with the help of a license key

Mnemonic	Opcode	Encoding
`iadd`	0x60	000
`isub`	0x64	001
`imul`	0x68	010
`idiv`	0x6C	011
`irem`	0x70	100
`ishl`	0x78	101
`ishr`	0x7A	110
`iushr`	0x7C	111
`iand`	0x7E	
`ior`	0x80	
`ixor`	0x82	

Table 1: JMark bytecode replacements for one equivalence class. Notice that the last three are not assigned a value. Two of these may be used to form another equivalence class to encode a single message bit, leaving one unused.

provided by the user when the program is installed. In this way, copies, legal or illegal, must be accompanied by the appropriate key. This key ties a copy of the software to the original purchaser for the purpose of providing information as to the source of discovered illegal copies.

Other static watermarking systems encode the data within the program text itself, taking advantage of the fact that instructions, functions, or other program units exhibit a high degree of independence. For any set, u, of program units that may be reordered, there are $|u|!$ possible arrangements. The United States patent issued to Davidson and Myhrvold [DM96] is one such system that reorders program units, in this case basic blocks. The original ordering of the basic blocks is recorded by the software producer and then modified in each watermarked copy. To expose the watermark in any copy, the ordering of basic blocks is observed by an analysis tool and compared to the original to recover the watermark data.

Venkatesan et. al. [VVS01] present what appears to be the strongest known static software watermarking technique. The idea is to treat the source program as a control flow graph G of basic blocks, to which a watermark graph W is added forming a new graph H. G and W are merged by adding code to the watermarked program that introduces new control flow edges between the two graphs. To detect the watermark the extractor needs to identify most of the nodes of W by considering the control flow graph of the watermarked program. The authors suggest to "store more one or more bits at a node that flags when a node is in W by using some padded data…". This appears to be a serious weakness of the algorithm. Even if an adversary does not have access to the exact method by which basic blocks are flagged as being members of W, he can apply a variety of local code optimization techniques (such as peephole optimization, register re-allocation, and instruction scheduling) that will completely restructure every basic block of the program. This will make watermark recognition virtually impossible.

Another static watermarking method is described by Stern et.al. [SHKQ99]. Like many media watermarking algorithms this one is based on spread spectrum techniques. The idea is to obtain a vector $c = (c_1, \ldots, c_n)$ representing the number of times a group of instructions i occurs in the original program. For example, c_5 might be the number of times the instruction sequence `mov %eax,%edx; push %eax` occurs in an x86 program. To embed the watermark the code is modified in such a way that in the watermarked program the instruction frequencies become $c \;=\; c + (w_1, \ldots, w_n)$. Typical modifications include swapping the order of data-independent instructions and replacing instruction sequences with equivalent ones. To detect the watermark we compute the instruction frequencies $d = (d_1, \ldots, d_n)$ of the watermarked program, and determine if c and d are ”similar enough” in which case we conclude that the program was watermarked. Unfortunately, it is easy for an attacker to perform the same sort of code modifications as the watermarker, effectively obliterating the mark. Furthermore, only one bit of watermark is embedded which is insufficient for most applications.

3.2 Dynamic Watermarks

Dynamic graph watermarking involves encoding the message text in a subgraph added to the heap. The watermark is detected by profiling the heap of a running program and searching for the subgraph. This technique was developed as an attempt to close some of the major attack strategies for static systems, namely the application of a semantics preserving program transformation, such as code optimization or obfuscation[4].

The process of watermarking a program using this method involves choosing an instance of a graph to encode the message text, and adding code to the program which, when executed, builds the graph in the heap. The code generation step is relatively simple, but the choice of the watermark graph is key. Since the watermark will be encoded as a subgraph in the heap, the problem of exposing the watermark is reduced to the subgraph isomorphism problem, which is known to be NP-complete [GJ79]. To keep the exposition computationally feasible, Collberg and Thomborson recommend using a family of enumerable graphs. A *family* is a set of graphs that have some structural properties in common. This aids in detecting the watermark in a heap profile because there may be some simple heuristics that identify subgraphs which exhibit the characteristic properties of the family.

We say that a family of graphs, G, is *enumerable* if there exists a function, $F : G \rightarrow \mathbb{N}$, and an inverse relation, F^{-1}. Notice that there may be more than one graph which maps to a specific

[4]Program obfuscation transforms programs to make the task of reverse engineering difficult [CTL97]

number, but for any graph in the family, it represents exactly one number. Many systems will use a family for which F is one-to-one, but this is not required for the purposes of watermarking. For the purpose of watermarking, the producer of the software picks a number $n \in \mathbb{N}$. Then graph g corresponding to n in the enumerable family G is picked, i.e., $g \in F^{-1}(n)$. Code is added to the program to embed graph g in the heap. For more information on counting and enumerating graphs and other combinatorial structures, see [GJ83] and [HP73].

There are two implementations of dynamic systems, both of which watermark Java programs, each using a different graph family. The SandMark system [CT99b] uses a radix-k graph to encode a number. A base, k, is chosen and the number is encoded in a circular linked list with one extra pointer per node. Each node in the list represents a power of k. If the extra pointer is null, then that node represents a coefficient of zero for the corresponding power. A self pointer encodes a value of one, and a pointer to any other node encodes a coefficient equal to the number of steps required to walk the list back to the node in question. A pointer to a single node in the list is kept which identifies the node representing k^n, where n is the highest power required to represent the data. Each successive node represents the next smaller power of k. An example of radix representation appears in Figure 2.

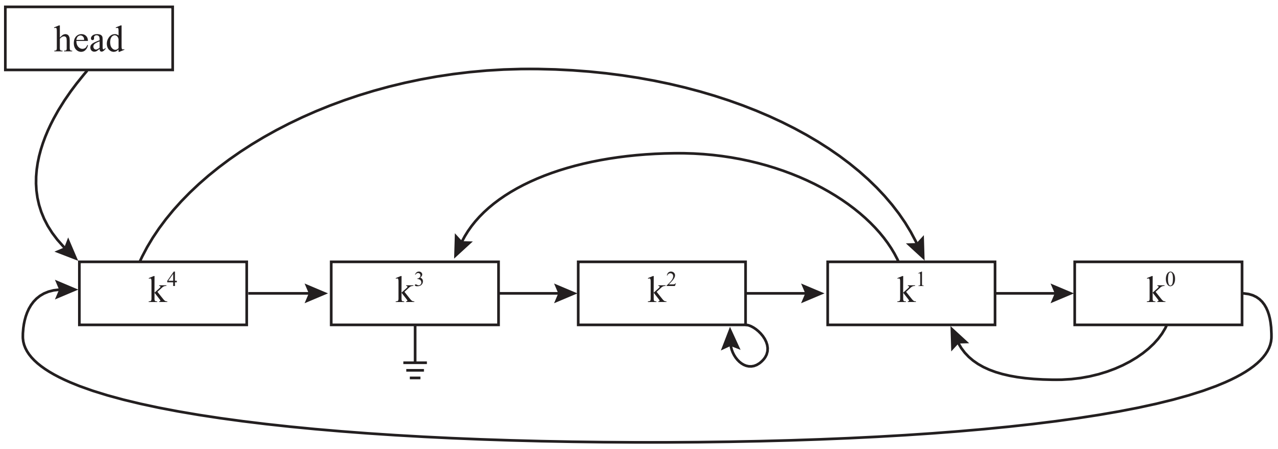

Figure 2: A radix-k graph. For $k = 10$, this represents decimal 30132.

SandMark also implements a feature by which the watermark only appears in the heap along a prescribed path through the program. In this way, the watermark behaves much like an Easter Egg found in some programs [NF98], although it is much more subtle because the normal user has no knowledge of the program's heap structure. Before the watermarking process begins, the developer instruments the target program with calls to profiling methods. When run with a prescribed input, a file is written with a trace of the instrumented program points. When the watermark code is inserted into the program, the calls to the profiling class are replaced with code which will incrementally build the graph.

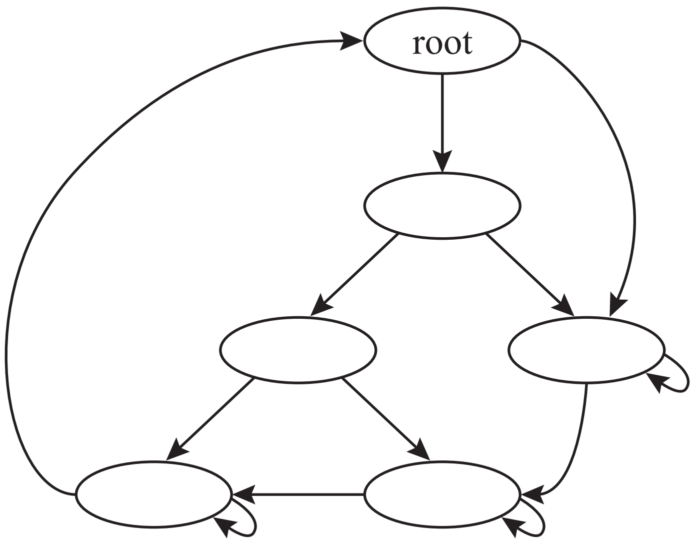

Figure 3: A planted place cubic tree with three leaves representing one.

The JavaWiz program [PKK$^+$00b] uses another family of graphs known as *planted plane cubic trees*, or *PPCT*'s [GJ83]. A PPCT is a binary tree that has a distinguished root. The root is connected in a circular linked list with each of the leaves, and each leaf has a self-pointer. An example of a PPCT is shown in Figure 3. Palsberg *et al.* [PKK$^+$00a] define a function, int, to map from PPCT's to integers as follows:

$$\begin{aligned}
\mathrm{int}(T) &= \mathrm{int}(T.\mathit{left}) \times \mathrm{c}(\mathrm{LeafNum}(T.\mathit{right}) + \mathrm{int}(T.\mathit{right}) \\
&\quad + \mathrm{min\_int}(\mathrm{LeafNum}(T.\mathit{left}),\ \mathrm{LeafNum}(T.\mathit{right})) \\
\mathrm{int}(leaf) &= 0 \\
\mathrm{min\_int}(L, R) &= \mathrm{min\_int}(L-1,\ R+1) + \mathrm{c}(L-1) \times \mathrm{c}(R+1) \\
\mathrm{min\_int}(1, R) &= 0 \\
\mathrm{c}(n) &= \frac{1}{n} \times \binom{2n-2}{n-1}
\end{aligned}$$

The $\mathrm{min\_int}$ function returns the minimum number that a binary tree represents with L and R leaves in the left and right subtrees, respectively. The function $\mathrm{c}(n)$ is the Catalan numbers [GJ83]. $\mathrm{LeafNum}(T)$ returns the number of leaves in its parameter tree. The left and right members of variable T are the left and right subtrees.

JavaWiz does not divide the graph building code over multiple program points, but instead inserts the code in one place, usually the `main` method of the target program. This makes locating and removing or modifying the graph building code very simple for an attacker armed with a decompiler. It should be noted that this system was implemented as a proof of concept, and not for practical use.

Collberg and Thomborson mention another graph encoding scheme that they refer to as an *enumeration* encoding [CT99a]. This encoding uses *parent pointer trees* and has not been implemented in any system. Although this family of graphs is enumerable, and therefore could be used to encode message texts, these graphs may not be practical. Since each node has only a single pointer to it's parent, locally this structure is similar to other very common graphs such as linked lists. Therefore, it may be very time consuming to locate a watermark of this type in a typical heap.

3.3 Other Anti-Piracy Techniques

Watermarking can only be effective against one threat to intellectual property, the use of illegal copies. It is not a viable tool in preventing industrial espionage. There is nothing a watermark can do if a competitor uses reverse engineering techniques to learn about the product and uses that knowledge to develop a competing product.

Function hiding, as developed by Sander and Tschudin [ST98], is an effective technique against such a threat, although it requires a very different business model. If a producer develops an algorithm which will provide a competitive advantage, the critical algorithm may be encrypted, such that it is not possible to reverse engineer the algorithm. When the encrypted algorithm is run on an input, the output is also encrypted, and requires that the output be sent to the producer of the software for the decryption step. Instead of selling the software, the producer of the product is selling the decryption service. Currently, their technique is only applicable to polynomials. Future work will be done attempting to generalize the methods so that they may be applied to arbitrary programs.

This technique eliminates the reverse engineering attack because the program does not contain the function, F, which is the trade secret. Instead it contains the encrypted function, $\mathcal{E}(F)$. With Sander and Tschudin's current technique for polynomials, it is still possible use reverse engineering to obtain the coefficients of $\mathcal{E}(F)$, but this is in effect the ciphertext. The plaintext coefficients are still difficult to obtain, due to the one way functions employed [MvOV97].

While this technique has practical limitations, it is a much stronger mechanism for the protection of intellectual property. Instead of being a steganographic technique, where an attacker has the ability to learn about the watermark by observing patterns in the cover text, this is a cryptographic technique. Therefore, the coefficients of $\mathcal{E}(F)$ will appear to be random and the process of learning about the encrypted information is made much more difficult.

This technique is only really viable when the computing resources required by the algorithm are significantly greater then the resources needed to decrypt the solution. If this condition is not met, then it would be more economical for the software service to be sold, and the customer would send the service provider the input along with payment. The producer or other service provider would run the program with the proprietary algorithm and send the results back. This is the model used by the developers of the MOSS [Aik94] system at the University of California at Berkeley. Again, this can be impractical because the software producer must also provide all of the computing resources required to process the workloads of all customers.

3.4 Media Watermarking

Although the goal of software watermarking is identical to watermarking other digital media, the methodology is quite different. When watermarking media, the data is hidden within the “noise” inherent in the medium that is imperceptible to the human eye or ear.

The simplest method for watermarking an image file is to replace the least significant bit of every pixel with one bit of the watermark data. This introduces an acceptable level of noise which the human eye should not be able to detect [KM92]. Unfortunately, this is also a very simple scheme to break by most any lossy compression algorithm, or by applying a simple filter.

A more sophisticated method for watermarking audio is *echo hiding* [GBL96]. Zeros and ones are encoded in the signal by adding a subtle echo to sounds which occur naturally. An echo at time δ_0 behind the original encodes a zero bit. A value of one is encoded with an echo at time $\delta_0 + \delta_1$. This system takes advantage of the fact that the human ear perceives an echo that is very close in time to the original as very mild distortion. The decoding process for this system consists of searching for the echoes and choosing likely values for δ_0 and δ_1. A second pass is made to find the sequence of bits which represents the message text. The inventors of this system tested it with some success against attacks using lossy compression algorithms. This system was successfully attacked by Petitcolas *et al.* [PAK98]. For more information on other steganographic techniques for audio and visual media, refer to the annotated bibliography by Petitcolas and Anderson [PA99].

4 Attacks on Software Watermarking Schemes

Petitcolas remarks that the difficult problem in watermarking is not in inserting the message texts, but in recognizing them later [PAK98]. This is certainly true, and the primary reason is the presence of the attacker. In order to prevent someone from detecting the watermark, the attacker is free to try anything to distort, destroy, or remove the watermark. The program implementing the exposition function $\mathcal{D}$ must be prepared for this possibility.

With a watermarked program, the attacker begins with the knowledge that it is a program, and any transformation applied must yield a program. The attacker will try to apply transformations that preserve the value, but just as the producer has an acceptable level of devaluation for the watermarking process, the attacker may be willing to accept further devaluation if the watermark can be removed, e.g., an attacker might be willing to tolerate a limited functionality for a program.

4.1 Information Gathering Attacks

Before beginning to make modifications to the program, the attacker may make an attempt to gather some information about the watermark. A first step may be to try and determine if the program is in fact watermarked, and if it is, which watermarking system was used.

The first and most obvious information gathering attack is to use the watermarking system itself. If the algorithms or tools implementing the algorithms are public, then the attacker may try to use the exposition tools of various systems to probe for watermarks. If any possible watermarks are found, the attacker now knows the system, and may be able to apply directed effort targeting a known weakness of that system.

If the producer has embedded a different watermark or serial number in each copy sold, there must be some difference between each pair of programs encountered. In this case, another simple information gathering attack may be carried out with a file comparison program, such as `diff`. This is an example of a *collusive* attack, since two or more watermarked copies are necessary. The comparison program may inform the attacker about the location of the watermark within the program by identifying the bytes in the executable which are different between the two copies. Knowing the location of the watermark allows for a more focused attack. For example, for a static system which encodes the watermark in an unused data segment, once the location is known, that data can be overwritten.

In addition to these simple information gathering attacks, almost any tool developed for the purpose of software engineering or program understanding may be employed. For example, a debugger may be useful for learning about the runtime behavior of the program. The attacker may use this to search for anomalous behavior that may indicate the presence of a watermark. For example, the program may allocate memory for a pointer-based data structure in a segment of code that does not seem to require such structures. Similarly, a profiling tool may be useful for searching for unusual program behavior, such as conditionals which always evaluate to the same value. These may be checks for error conditions, but also may be *opaque predicates* [5] protecting some aspect of the watermark.

Potentially the most powerful tools in an attacker's arsenal are the decompiler or disassembler. If assembly or even source code can be obtained, then the attacker has a powerful and convenient platform for making and testing modifications to the program in a more easily understandable form.

[5]An *opaque predicate* is an expression which always evaluates to the same value at runtime, but the aforementioned property of the expression cannot be discovered by standard static analysis techniques. The concept of opaque predicates was introduced in [CTL98].

For many languages, such as C and C++, decompilation may be very difficult, and the attacker may be forced to analyze binaries. However, there are very effective decompilers available for Java bytecode, such as [Kou01], and [vV96]. This alone may make secure watermarking of Java programs very difficult.

4.2 Program Transformation Attacks

There are three basic methods by which an attacker may attempt to foil the watermark exposition program. The watermark can be cut out completely by removing the code or data that comprises the message text. This is a *subtractive* attack. Another approach is to add other watermarks or useless data in an attempt to confuse the exposition program and prevent it from locating the authentic watermark. This is an *additive* attack. Finally, a *distortive* attack may succeed in mangling the watermark so that it is unrecognizable, or so that it now encodes a different message text.

A specific transformation which embodies one of these strategies may fall into one of the two categories, *conservative* and *speculative*. A conservative attack is one that preserves the semantics of the program. We call it conservative because it is always safe to apply because the resulting program will be functionally identical. Program optimization and obfuscation are two of the interesting and potentially most powerful conservative attacks. Optimizations, such as code scheduling, function inlining, and basic block layout modifications may disrupt the patterns that the exposition program relies on to locate watermarks. Obfuscation may also change the control flow, but also may split types, promote variables, and complicate the heap with extra pointer fields (see [CT99a] and [CTL97] for examples). An example of obfuscation transformation on heaps is shown in Figure 4.

A slightly weaker transformation that might still be effective against some systems is the use of a decompiler or disassembler and immediate recompilation. When regenerating the executable, the compiler may make different choices of instructions, code layout, or other features on which an exposition program might rely on [MIM$^+$00].

Another conservative transformation is the addition of new watermarks into the program. This attack may succeed on two levels. First, the new watermark may distort the original or it may in some other way confuse the exposition function. In this way it is acting as an additive attack. Second, in the case that the original watermark is left intact, the appearance of a second watermark may cast into doubt the identity of the producer.

A speculative transformation is one that risks damaging the functionality of the program, possibly to the point where it is unusable. Of course, the attacker may attempt multiple such transformations iteratively until a more desirable result (for the attacker) is produced. Most speculative attacks can be greatly assisted by some prior knowledge. The potentially most effective attack is a *complete subtractive attack*. If the attacker can learn where the watermark is and how it is en-

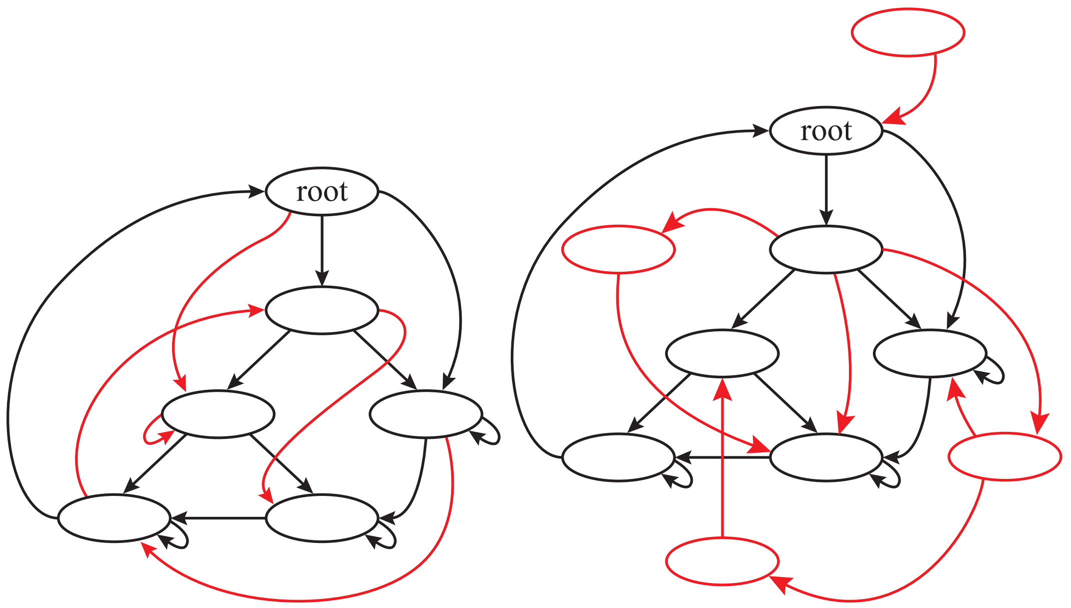

Figure 4: Semantics preserving obfuscating transformations targeting the heap. On the left, only pointers are added. On the right, extra nodes further complicate the task of the exposition function.

coded, then the attacker can completely remove it. In the process, it is possible that some aspect of the program other than the watermark may be disrupted, but repeated attempts using information gathered at each stage may be successful.

Another powerful speculative attack may be launched on a dynamic graph watermark if the graph can be located in memory. If found, then the attacker may insert extra code which changes some of the pointers. This attack may either destroy the watermark by changing the structure expected by the exposition program, or preserve the structure and change the encoded value. In either case, the intended watermark cannot be recovered.

4.3 Attacks on Static Systems

Holmes' watermarking system [Hol94] which encodes the message text in an unused data segment may be the simplest system to break. Since the watermark data is in the same location in each copy of the code, a collusive attack using a file comparison program may be used locate the watermark. Once the location is known, it is trivial to overwrite that data. Since this system does not make any tamper-proofing efforts, this attack is efficient and effective.

Recall that Samson [Sam94] embeds three integers that exhibit an unusual property in the data segment. Moreover, an additional method is added to the program to check the property of the embedded watermark. An attack on Samson's system may begin with a collusive attack identical to the attack on Holmes. Once the data is overwritten, however, the program is "broken" because the watermark validity check will likely fail. However, to a sophisticated attacker, the failure of the program may provide clues to drive further transformations. Using a debugger, the attacker may learn exactly where the failing validity check is located in the code. Removing the check may be trivial using an instrumentation library such as Dyninst [BH00]. The Dyninst API is a set of functions which support attaching to running programs, inserting code into those programs, as well is replacing or deleting code. This library is used for legitimate purposes to develop profiling and debugging tools. Miller *et al.* [MCI$^+$] describe a method using Dyninst for removing a similar program validity check, a call to a function that contacts a license server from a commercial program.

Since we could not locate an implementation of Holmes' and Samson's methods are patented, we have not been able to experiment with their techniques and test these attacks. Monden's methods, described in [MIM$^+$00], are implemented in a tool called JMark [Mon00], which is freely available for download. Recall that Monden et al. embed the watermark by adding extra methods. The authors tested their technique on two attack methods with some success. On each of ten watermarked classfiles, they applied the SourceGuard obfuscater [Inc01]. After obfuscation, the JMark exposition tool was successful in all cases. Secondly, they decompiled each classfile with `Mocha` [vV96] and recompiled with `javac` [Mic01]. Five of the classfiles failed to decompile, and the five remaining had a total of eight watermarks. Five of these were successfully exposed after recompilation. While these results are promising, the authors did not test their system against an optimization attack. Since many optimizations either reorder code or replace operands, it is obvious that such an attack could be very successful. To test this hypothesis, we used the BLOAT [HN99] Java optimizer on classfiles that were watermarked with JMark. *This attack was successful on every attempt.*

Moskowitz' [MC98] system is patented and we were not able to find an implementation of his methods, we do not have access to Therefore, we can only hypothesize about potential attacks. Because the message text is part of the original program, we cannot launch an attack which attempts to remove or disrupt the watermark without destroying the program. However, with some concerted effort, it may be possible to obtain a snapshot of the state of the program after the code has been extracted and use this information to reconstruct a complete binary. This may be technically very difficult, but it may be possible for a highly sophisticated attacker.

4.4 Attacks On Dynamic Systems

Realizing that many static systems could be easily defeated by many semantics preserving transformations, Collberg and Thomborson set out to develop a new system which would resist such automatic attacks. With a dynamic graph technique [CT99a], the watermark is built by code inserted by the watermarking tool. If the program is modified but the semantics are preserved, the heap behavior, and thus the watermark, must also be preserved.

Because JavaWiz and SandMark are readily available for download, we were able to experiment with each. To set up the experiments, we had a colleague watermark a moderately large Java programs and obfuscate the compiled classfiles with WingGuard [Cor00]. We began our attack attempts with no knowledge of the source code, and armed with only a decompiler, an editor, the watermarking tools, and a compiler. All experiments were carried out on a Intel Pentium III 700 MHz computer, with 128MB of physical memory, running Red Hat Linux kernel version 2.2.19-6.2.1 and Sun's Java Virtual Machine version 1.2.2-L.

4.4.1 Attacking JavaWiz

Since most of the code inserted by JavaWiz is in the `main` method of the program or somewhere else along the program's initialization path, it very easy to locate and remove. We ignore this attack for JavaWiz because a serious implementation of this system would attempt to disguise the inserted code through program obfuscation or other techniques. The authors tested JavaWiz against two semantics preserving attacks, obfuscation and optimization [PKK$^+$00a]. In their testing, all attacks were unsuccessful. We tested JavaWiz against three additional attacks, two conservative and one speculative.

We attempted an additive heap attack, which obfuscates the heap by inserting extra nodes. A class in the program is selected, which has two members which are references to other objects of the same class. We allocated a collection of these nodes and randomly connected them into a large jumbled graph. We tried adding various numbers of extra nodes, and in all cases, the exposition function was still successful. Results from additive attacks on JavaWiz are shown in Figure 2.

Nodes Added	Exposition	Time
50	success	19.120s
100	success	19.230s
1000	success	20.600s
10000	success	42.610s
50000	success	168.810s

Table 2: Results from an additive heap attack on JavaWiz. Success indicates that watermark exposition was successful. Time is measured in seconds.

Next, we added extra pointer fields to various classes and randomly assigned values to them. This attack complicates the task of the exposition function because the degree of many nodes in the graph is increased, and there may be more candidate classes which have the requisite two reference fields to be considered a potential PPCT node class. We tried adding one, two, and three extra pointer fields, but were not able to cause the exposition function to fail. Additionally, the execution time for each exposition attempt was minimally impacted.

Finally, we tried a speculative attack. First, information was gathered about the watermarking tool by testing it on an `''Hello World''` program. We examined the modified code and determined what sort of code additions we could expect to find in the target program. Specifically, we searched for what types of fields were added to what classes, and what methods were added or modified. On our simple example, we quickly discovered that three fields and one method were added to the class being used as a vertex in the PPCT.

The target program was decompiled using Jad [Kou01] and using the information gathered, the node class and the PPCT edge fields were located. We inserted code which modified the edge fields in a few randomly selected nodes, recompiled and ran the exposition program. *No watermark could be located after this transformation.*

4.4.2 Attacking SandMark

SandMark [CT99b] was tested against these same attacks. The attack adding nodes to the heap was not successful even in slowing down the exposition program. The second attack adding extra edge fields was equally unsuccessful.

Using a speculative strategy identical to that used for JavaWiz, we randomly modified a single edge field in one node of the graph by adding one line of code in the `main` method of the program, recompiled, and tried to detect the watermark. *As was the case with JavaWiz, the watermark was not located.*

5 UWStego: an extensible architecture for dynamic software watermarking

Since dynamic watermarking systems are resilient to several attacks, we believe that they merit further investigation. To aid in this future work, we are in the process of developing the `UWStego` toolset. We have designed this toolset with the primary considerations of *portability, extensibility*, and *resilience*.

Portability: Portability is achieved by implementing this system in the Java programming language [Mic01] using only standard classes.

Extensibility: Extensibility is achieved with an architecture that allows any Java programmer to implement new message text encoding systems, code generators, and watermark exposition systems. `UWStego` provides an abstract interface for encoding and exposition of watermarks. Therefore, if a designer devises a new enumerable family of graphs, they can easily incorporate

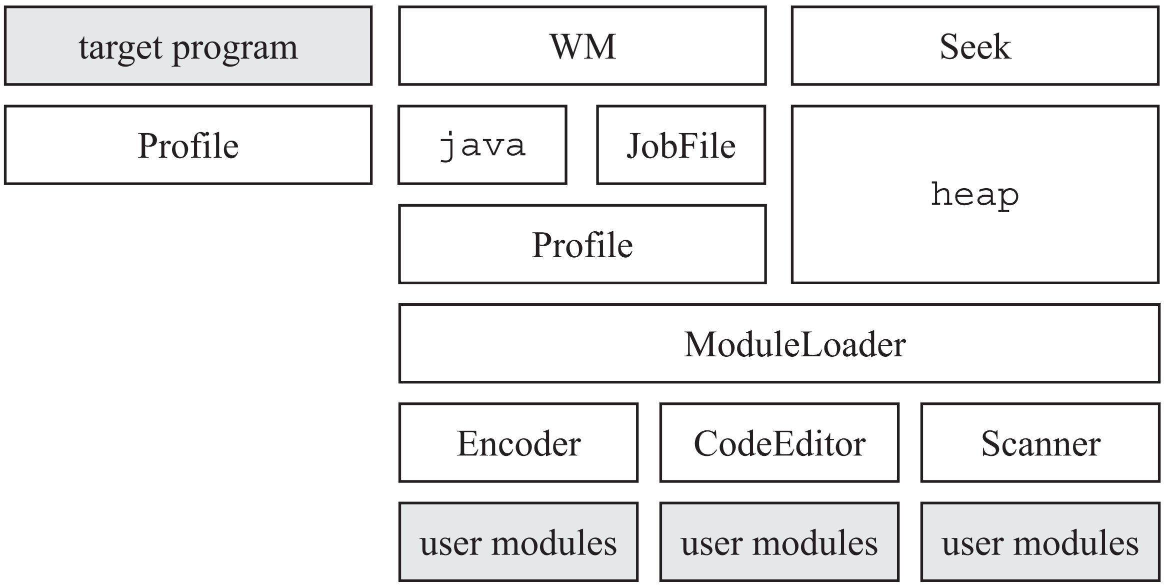

Figure 5: `UWStego` toolset architectural overview. Grayed boxes indicates that the user provides or specifies this component. The `java` and `heap` packages provide classes for loading and manipulating Java classes and heap profiles, respectively.

it into `UWStego` modules. Third-party users may take advantage of our Java classfile editing and heap profile analysis systems.

Resilience: Suppose a dynamic software watermarking system uses a single family of enumerable graphs, e.g., PPCT. In this case, an attacker can target their attack to this specific family. On the other hand, imagine that we have a library of families of enumerable graphs. In this case the specific family of graphs used for watermarking can be the vendor's secret. Since an attacker does not know the specific family of graphs that the vendor uses, the resilience of the software watermarking system is enhanced. Collberg and Thomborson suggest that a watermarking tool should have a library of many encoding technique [CT99a]. It is easy to incorporate new families of enumerable graphs in `UWStego` and therefore our toolset facilitates creating a library of families of enumerable graphs.

An implementation of a watermarking system is a specialized program which makes modifications to another. The `UWStego` toolset takes advantage of this fact by structuring most of the code and heap manipulation features in such a way that they can be used to develop many different watermarking systems. The `UWStego` toolset is specifically designed for implementations of dynamic graph watermarking systems, but it can be modified for use with many other types of systems. Architectural view of the `UWStego` architecture is shown in Figure 5.

5.1 Generic Dynamic Watermarking Algorithms

In this section we describe generic algorithms for dynamic software watermarking. These generic algorithms provide an abstract description of encoding and exposition functions, and form a basis for the abstract programmer interface that allows a programmer to "plug in" an enumerable family of graphs.

To embed a dynamic graph watermark, $w \in \mathcal{W}$, in a program, $p \in \mathbb{P}$, we must also choose an enumerable family of graphs and a method of generating the code to build the desired graph in the heap at run time. Consider $\mathbb{S}$ to be the set of enumerable graph families that may be used as watermarks, and $\mathbb{E}$ to be the set of algorithms for generating the code for constructing graphs. The generic encoding function $\mathcal{E}$ is shown in Figure 6.

The ENCODEMESSAGETEXT function is selected based on the specified graph family, f. It takes the message text, w, as a parameter and returns the watermarked graph. The GENERATE-CODE function is selected by the c parameter. It takes the target program, p, and the graph, g, from the previous step and outputs the watermarked program. For example, in case of JavaWiz f is the family of PPCTs and g is the PPCT corresponding to w. Moreover, the watermarked program p_w contains the code for adding the PPCT to program p.

```
   DYNAMICGRAPHWATERMARK($p \in \mathbb{P}, w \in \mathcal{W}, f \in \mathbb{S}, c \in \mathbb{E}$)
1      $g \leftarrow$ ENCODEMESSAGETEXT$_f(w)$
2      $p_w \leftarrow$ GENERATECODE$_c(p, g)$
```

Figure 6: The Abstract Encoding Algorithm

A generic exposition function is shown in Figure 7. To expose a dynamic graph watermark in a program, the DYNAMICGRAPHEXPOSE function needs the state of the program, $S(p, I, t)$, and must also know the graph family f to be used by the watermarking system. Additionally, this function must know how exactly the graph is built, because various code generating algorithms might make use of different programming language constructs in implementing the watermark.

The RECONSTRUCTSTATE function takes the observed program state, $S(p, I, t)$ and constructs a graph representation that the rest of the function can use. Depending on the language of the target program, this representation may be annotated with addition information, such as the type of each object. For the Java programs the `UWStego` toolset was designed to work with, this type information is available.

After reconstructing the state of the program, each node in the heap is examined to determine if it is part of a potential watermark. Depending on the graph family and the code generating algorithm, the ISPOTENTIALWATERMARKNODE function uses heuristics to narrow down the possible watermarks to just a few. For clarification, we will will gain take the example of JavaWiz. Recall that JavaWiz uses PPCTs as its family of graphs. The root node of a PPCT has one incoming edge and two outgoing edges, and is start of a cycle (see Figure 3). This characteristic of the root node of a PPCT can be used in the implementation of IsPotentialWatermarkNode.

If a node, n, is identified as a potential watermark, the IsValidWatermark function determines if a graph rooted at n is actually an instance of a graph of the correct family. If the graph does have the correct structure, it's value is found by the WatermarkValue function and this value is added to the set of exposed watermarks. After searching all nodes, n, in the heap, the set of exposed watermarks is returned.

```
DynamicGraphExpose($S(p, I, t), f \in \mathbb{S}, c \in \mathbb{E}$)
1     $heap \leftarrow$ ReconstructState($S(p, I, t)$)
2     $found \leftarrow \emptyset$
3         for each $n \in heap_v$
4             if IsPotentialWatermarkNode$_{fc}(n)$ then
5                 if IsValidWatermark$_{fc}(n)$ then
6                     $found \leftarrow found \cup$ WatermarkValue$_{fc}(n)$
7                 end if
8             end if
9         end for each
10    return($found$)
```

Figure 7: The Abstract Exposition Algorithm

5.2 User Interface

The toolset we have implemented has a simple command line interface which consists of two programs and one Java class. The system is divided into three steps: *profiling, watermark insertion, and watermark exposition*. The user interfaces corresponding to the three steps are defined in the `edu.wisc.cs.stego` package.

5.2.1 Program Profiling

To increase the stealth of the watermark, a good watermark embedding system (for any dynamic message text encoding) should allow the watermark to appear along only one execution path through the program. This feature was included in the SandMark [CT99b] system, and has also been included in the `UWStego` toolset in the `Profile` class.

To use this feature in watermarking a target program, the user must select a *watermark exposition input*. The watermark exposition input consists of an ordered set of input files and/or user interactions. This input should be chosen in such a way that the path through the program on this input is completely deterministic, i.e., no point along this path, should the call to a method in the `Profile` class, should be dependent on something such as the amount of memory or the type of network connection the host computer has. [6] This is necessary so that the watermark can be demonstrated on any computer that has an appropriate Java runtime environment.

The `Profile` class provides two methods which are used to gather information about the target program. The `Profile.initializeProfile` method should be called from some point along the startup or initialization path through the program. This must be called along all paths from the start of the program to any call to the `Profile.recordPoint` method on any input. When the program is being profiled, the `initializeProfile` method takes a single argument which is an instance of the `java.lang.String` class which contains the name of a file to be written with the profile data. When this method returns, the specified file is opened and the location in the program is recorded in the file.

After the profile is initialized, the `Profile.recordPoint` method makes note of the location of the program in the file. Calls to this method should be placed at various program points which will be *passed* on the watermark exposition input. It is along this path through the program that the watermark will appear. After instrumenting the program with calls to the methods in the `Profile` class, the program should be run on the watermark exposition input to generate a profile which will be used in the watermark insertion phase.

5.2.2 Watermark Insertion

The `WM` program performs the watermark insertion phase. It reads the profile generated in the previous phase and locates the calls to the profiling methods in the bytecode. The calls to those methods will be replaced with code that builds the watermark graph in the heap.

The `WM` program requires a single command line argument which is the name of a *job file*. The job file specifies the parameters required for watermarking a Java program and is an ASCII text file containing a number of lines, each of which provides a value for a single parameter. Each line begins with a parameter keyword, with white space separating the keyword from the data. Some parameters are allowed to contain spaces, and these must be delimited by [: and :]. The order of the parameters in the job file are unimportant to the `WM` program, but may be important to any modules. The arguments to the `Encoder` and `CodeEditor` implementations are passed in the

[6] Allocation of a reasonable number of objects need not be considered non-deterministic, although it is somewhat dependent on the runtime environment. It must be assumed that the watermark detection environment meets some basic set of requirements.

order encountered in the job file. A description of the various fields that appear in the job file is given in Table 3, and an example of a job file is given in Figure 4.

Keyword	Enclosed	Required	Description
`data`	yes	yes	the text to be encoded in the watermark
`profile`	no	yes	the name of the file containing the profile data
`target`	no	yes	the path to the directory containing the target program
`outdir`	no	yes	the path to the directory where the modified classfiles should be placed
`node`	no	yes	the name of the class which should be used as the node class in the graph
`encoding`	no	no	the fully qualified name of the class to use for data encoding
`e-arg`	yes	no	an argument to the encoding class
`ceditor`	no	no	the fully qualified name of the class to use for bytecode modifications
`ce-arg`	yes	no	an argument to the code writer class

Table 3: Descriptions of entries in the job file.

5.2.3 Watermark Exposition

The `Seek` program carries out the watermark detection phase, with arguments specified on the command line. Two command line arguments are necessary, the name of the `Scanner` implementation, and the name of a file containing the heap profile. The profile is generated by running the target program on the watermark exposition input with the `-Xrunhprof` option of the Java Virtual Machine [Mic01]. This file is parsed and the state of the heap when the program finishes is recreated. The `Seek` program then scans the heap for potential watermarks, writing the results back to the console.

5.3 Programmer's Interface

In order to aid in future watermarking research, we have designed our toolset to be extensible. The basic toolset provides the framework for reading and writing classfiles, reading heap profiles, as well as the basic utility work of parsing arguments and job files. The toolset allows any user to write new modules, which integrate seamlessly to allow experimentation with different data encoding schemes.

```
data [:Copyright 2001 ABC Corp.:]
profile test/test.prof
target .
outdir ./testwm
node test.Main
encoding edu.wisc.cs.stego.modules.EncoderLoadTest
e-arg [:argument to EncoderLoadTest:]
ceditor edu.wisc.cs.stego.modules.Unprofile
ce-arg [:argument to Unprofile:]
```

Table 4: Example job file.

In order to implement a new encoding scheme, a programmer implement write three Java classes, each of which inherits from a different abstract class provided in the `edu.wisc.cs.stego` package. The toolset uses the `java.lang.reflect` classes along with the virtual machine's class loader system to instantiate the module classes. After obtaining an instance of a module specified by the user, the toolset interacts with each module through the methods defined in the parent abstract classes. There are three abstract classes which correspond to generating the graph to be used for watermarking, encoding the graph in the heap, and exposing the graph in the heap respectively.

To simplify loading and instantiating module classes, each object is required to implement a factory method. This factory method should use the `new` operator to create the requested object and return a reference. The factory methods each take an array of strings. For the `Encoder` and `CodeEditor` classes, the strings are read in from the job file. For the `Scanner` class, they are specified on the command line of the `Seek` program.

The `edu.wisc.cs.stego.Encoder` class defines the prototype for a method for modules which generates a graph from the watermark data. Any class which inherits from the `Encoder` class must implement two methods:

- `public static Encoder buildEncoder(String[] argv)`
- `public edu.wisc.cs.util.WatermarkGraph encodeWMData(String data)`

The `buildEncoder` method is the factory method. The factory methods for the other two module classes follow the same pattern, all taking an array of parameter strings and returning the constructed object.

The `encodeWMData` method will be called by the `WM` program to obtain a representation for the watermark graph. The `data` parameter will contain the character data to be encoded within the graph.

After obtaining the watermark graph, the `WM` program uses an instance of a class which inherits from `edu.wisc.cs.stego.CodeEditor` to add the bytecode necessary to build the graph in the runtime heap.

- `public static CodeEditor buildCodeEditor(String[] argv)`
- `public void instrumentClasses(WatermarkGraph graph, ProfilePoint[] profile, HashMap classes, String nodec)`

The `instrumentClasses` method does the work of removing the calls to the profiling methods and replaces them with code that will build the watermark at runtime. To hide the user from the details of manipulating the Java bytecode, we have implemented a `GenericEditor` class which can instrument a program with code to build any graph that can be represented with the `WatermarkGraph` class. This implementation may not be as stealthy as possible, but it frees the watermarking researcher to concentrate on the more difficult problem of watermark resilience and exposition. If a scheme is found to warrant further experimentation, it would be advisable to implement a custom `CodeEditor` specific to the graph family being use. This would allow the code generated to be the best possible for a given graph family and result in a more stealthy watermark.

- `public static Scanner buildScanner(String[] argv)`
- `public String[] scanHeap(HeapProfile hp)`

The `scanHeap` method searches the heap for potential watermarks and returns an array containing any potential message texts found.

6 Future Work

The field of software watermarking is relatively new. Techniques that are currently being investigated have several weaknesses and thus the area of software watermarking provides a great deal of opportunity for further research.

6.1 Dynamic Watermarking

Since most dynamic techniques are resilient to several semantics preserving transformations, we believe that they merit further investigation. We see two areas which need substantial improvement, the types of graphs used, and the code insertion mechanism.

What makes radix-k and PPCT watermarks easy to break is that they are simple to locate. Since they are in some ways unusual, there is a good chance that such a structure is a watermark, if one is found. We believe that it is the regular structure of these graphs that enables the graph scanning heuristics to find them so efficiently. It would be interesting if a more general graph was used to encode the watermark, especially one that does not have such a regular structure and resembles the heap structure of the program being watermarked.

To make such a graph useful for watermarking, an algorithm must be developed which would allow efficient exposition of the graph on demand to someone with a secret key. As in cryptography, discovery of the watermark graph should be very difficult to anyone without the key.

To solve this problem, we are currently investigating a result from the study of random graphs. It has been shown that the subgraph isomorphism problem can be solved in polynomial time for a random subgraph if the degree sequences are known [Lip78]. For example, nodes in a directed graph can be classified by their in-degree and out-degree. Degree sequences are a generalization of this concept. We are investigating whether degree sequences can provide a stealthier technique for dynamic watermarking of programs.

If it is now difficult for an attacker to detect the watermark graph, it still may be relatively simple to detect the code which builds the graph by decompilation and inspection. To resolve this, we believe much work needs to be done to improve the stealth of the watermark code, or the *static stealth*. The simplest method improving the static stealth is in the choice of watermarking systems. If a large library of systems can be developed, the programmer may choose the system based on the traits of the target program and the code inserted by the type of code inserted by the system. By matching the programming idioms used, it will be more difficult for an attacker to differentiate the program code from the watermark code. Our package `UWStego` facilitates the construction of libraries of families of enumerable graphs.

All of the current work on dynamic software watermarking is focused on heap embedded watermarks. Another direction we see for the future is searching for other types of dynamic systems. We believe Collberg and Thomborson's heap embedded graph idea has merit, but there are many other opportunities to be explored. For example, an extra parameter can be added to some methods in the program which are occasionally called in a sequence. Code can be added to each of these methods that performs some simple operations. The results obtained in each method are passed along to the next. After a specific sequence of calls to the modified methods, the watermark is built and temporarily stored in some global variable. At other points in the program, this global variable is cleared and potentially used for other purposes. In this way, the existence of the watermark is stealthy at runtime because of it's locality to particular methods, and the fact that it will be destroyed soon after it appears. It can also be made statically stealthy if the computation at each stage is similar to other computation in the method. Additionally, the watermark computation can be keyed on values from the normal operation of the program. To expose this watermark, the program must be run on an input which causes the watermark path to be executed. As it is run, the value of the global variable is watched by a debugger or another similar program.

6.2 The `UWStego` Toolset

Currently, the `-Xrunhprof` option on the Java Virtual Machine is not really a profile of the heap but a snapshot of the heap just after the program has finished executing. For watermarking, this means that any dynamic graph structure which is embedded must be referred to by a `static` data members. Any object which is not accessible through a static member will already have been reclaimed by the garbage collector by the time the profile is recorded. Using this current technology greatly reduces the stealth and resilience of a watermark, because once the watermark is created in memory, it must be retained for the duration of the program. This gives the attacker a larger window of opportunity in which to observe and tamper with the watermark.

A logical next step is to create a watermark exposition environment that writes a heap profile when some user specified condition is met. Using such a system would allow the watermark structure to be created and destroyed in a short span in the program. When attempting to expose a watermark, the producer would specify a predicate. The runtime environment would watch the value of the predicate in the program and write the heap profile when it evaluates to true.

Additionally, we would like to develop a library of modules that implement different watermark encoding schemes, along with corresponding exposition modules. Even within the dynamic graph watermarking paradigm, we believe there are numerous possibilities for families of graphs that may be used as watermark encodings which have yet to be explored. This library will give researchers a more diverse range of techniques with which to experiment, and, in a practical sense, also makes the job of an attacker more difficult.

6.3 Watermarking with Other Languages

Thus far, all of the interesting watermarking systems implemented have targeted programs written in Java. From a research perspective, this is useful, because both the language itself and the bytecode format are simple to work with. For experimental purposes, research ideas are relatively easy to implement in a real system.

Conversely, it is quite unrealistic to expect a Java watermarking system to be satisfactorily secure because the bytecode distribution format retains much of the information contained in the source code. Our experience shows that unless tools are developed implementing much stronger obfuscating technology, speculative attacks involving decompilation and manual modification have a high likelihood of succeeding. Perhaps obfuscation techniques such as those discussed by Collberg, Thomborson, and Low [CTL97] will ameliorate this weakness.

With improved obfuscation techniques, Java watermarking may be commercially viable, but currently, much software is still written in languages such as C and C++. Therefore, it would be useful to take a serious look at watermarking these languages. It may be considerably more difficult to edit the code at the binary executable level, but may be reasonable at the level of the intermediate form of the compiler. Regardless of the level at which the watermark is inserted, the difficulty of the analysis and successful modification of a natively compiled binary will greatly increase the security of the system.

6.4 Limits of Software Watermarking

Developing improvements to current software watermarking technology presents a significant challenge. We believe that this is due to the fact that the watermarked product is still a program, and as such, it is information rich. Furthermore, there are numerous software engineering tools which have been developed for the purpose of aiding engineers in learning about a program, e.g., debuggers, profilers, disassemblers, and decompilers. These tools are every bit as useful for an attacker as they are for a legitimate engineer. Attacks on dynamic systems may use some reverse engineering techniques to locate the watermark generating code, which then may be removed. Profiling may be used to locate opaque predicates. Disassemblers and decompilers may be used to transform the program into a more understandable format, allowing for easy editing.

Since an attacker can be expected to use all of the available technology, it is the task of the watermark designer to take these tools into account and prepare for them as best as possible. Knowing this, the most important question to be asked is, in the presence of sophisticated attackers, what is the best any software watermarking scheme can do?

To answer this question, it first must be determined if it is possible to model speculative attacks. Since these attacks are not algorithmic and very much programmer driven, this may be very difficult or impossible.

6.5 The Ideal Software Identification System

There is no doubt that anyone in the profession of software design and development considers it to be as much art as it is engineering. Two teams of programmers, given the same software specification, are certain to make different design decisions on nearly every level, from program architecture to the code that implements it. In fact Knight and Leveson [KL86] demonstrate the two different teams of software engineers can produce significantly different design and code even when they work from the same specification.

The ideal software identification system would capitalize on this fact. Instead of using a watermark to add a signature, it would extract a fingerprint of the program which is intrinsic to the design in much the same way that a person's fingerprint is intrinsic to a person. A naive method for fingerprint extraction would be to observe the order of call stack pushes and pops. The primary difficulty in developing such a system is to assure that the signature is preserved when program transformations are applied. For example, our naive system is easily defeated by applying function inlining and outlining.

If a software fingerprinting system can be developed which is resilient to semantics preserving transformations, it may be the ideal identification system. If no modification is made to the program, an attacker has no way to determine whether or not the program is being protected with any sort of identification system. If no decision can be made, the attacker intent on making illegal copies has three options. First, the illegal copies can be made without any modification. In this case those copies can be trivially identified by a fingerprint extraction system. Second, the attacker may apply semantics preserving transformations in hopes that this will remove or distort any watermark or fingerprint. This is the situation that the fingerprinting system designer must be most concerned with. Lastly, the attacker may attempt some speculative program transformation which is not semantics preserving. If this action is taken, the program is "broken" and the value of the copies distributed is diminished because it is not functionally equivalent to the original.

7 Conclusion

The field of software watermarking is in its infancy. The systems proposed are interesting and exhibit the range of diversity possible in the field, but many have vulnerabilities which are easy to exploit. Therefore, the field of software watermarking is not yet commercially viable as a strong deterrent to intellectual property theft and software piracy.

In this paper we presented metrics for evaluating software watermarking systems. We also presented an extensible architecture `UWStego` for watermarking JAVA programs. `UWStego` is a dynamic watermarking scheme and allows a software engineer to design new graph encoding schemes and easily incorporate it into the architecture. We also discussed how the resilience of a software watermarking scheme is enhanced by having a large library of encoding schemes, which is facilitated by `UwStego`.

The next step for the watermarking technology is in characterizing speculative attacks and developing systems that resist them. The most promising strategy may be increasing the stealth of the watermark. If an attacker has difficulty learning about the watermark's structure, it is much harder to break. It may be possible to improve the stealth of dynamic watermarking techniques. If this can be done, this may be the breakthrough in the software watermarking field, because most dynamic techniques are already immune to most semantics preserving attacks. To aid the research in this area we are developing the `UWStego` toolset which is structured for easy experimentation with dynamic watermarking techniques.

References

[Aik94] Alex Aiken. MOSS: A system for detecting software plagiarism. `http://www.cs.berkeley.edu/~aiken/moss.html`, 1994.

[BH00] Bryan Buck and Jeffrey Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[Cor00] WingSoft Corporation. Introduction to WingGuard 2.0. `http://www.wingsoft.com/wingguard.html`, 2000.

[CT99a] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Symposium on Principles of Programming Languages (POPL)*, pages 311–324, 1999.

[CT99b] Christian Collberg and Gregg Townsend. SandMark: Software watermarking for java, 1999.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, 1997.

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *In symposium on Principles of Programming Languages (POPL)*, San Diego, CA, January 1998.

[DM96] Robert Davidson and Nathan Myhrvold. A method and system for generating and auditing a signature for a computer program. US Patent 5,559,884, Assignee: Microsoft Corporation, September 1996.

[FP98] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Revised*. PWS Publishing Company, 1998.

[GBL96] Daniel Gruhl, Walter Bender, and Anthony Lu. Echo hiding. In *Information Hiding: First International Workshop*, pages 295–315, Berlin, Germany, May 1996. Springer-Verlag.

[GJ79] Michael Garey and David Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, New York, 1979.

[GJ83] Ian P. Goulden and David M. Jackson. *Combinatorial Enumeration*. John Wiley, New York, 1983.

[Gla00] Jeff Glasser. The Software Sopranos, 2000. US News and World Report.

[HN99] Anthony Hosking and Nathan Nystrom. BLOAT: Bytecode Level Optimizer and Analysis Tool. `http://www.cs.purdue.edu/homes/hosking/bloat/`, 1999.

[Hol94] Keith Holmes. Computer software protection. US Patent 5,287,407, Assignee: International Buisness Machines, February 1994.

[HP73] Frank Harary and Edgar Palmer. *Graphical Enumeration*. Academic Press, New York, New York, 1973.

[Inc01] 4^{th} Pass Inc. SourceGuard. `http://www.4thpass.com/sourceguard/index.html`, 2001.

[KL86] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions of Software Engineering*, 12(1):96–109, 1986.

[KM92] Charles Kurak and John McHugh. A cautionary note on image downgrading. In *Computer Security Applications Conference*, pages 153–159, 1992.

[Kou01] Pavel Kouznetsov. Jad: The fast JAva Decompiler. `http://www.geocities.com/SiliconValley/Bridge/8617/jad.html`, 2001.

[Lip78] R. Lipton. The beacon set approach to graph isomorphism, 1978. Yale Dept. Comp. Sci. preprint No. 135, 1978.

[MC98] Scott Moskowitz and Marc Cooperman. Method for stega-cypher protection of computer code. US Patent 5,745,569, Assignee: The Dice Company, April 1998.

[MCI$^+$] Barton Miller, Mihai Christodorescu, Robert Iverson, Tevfik Kosar, Alexander Mirgorodskii, and Florentina Popovici. Playing inside the black box: Using dynamic instrumentation to create security holes. Submitted for publication, February, 2001.

[Mic01] Sun Microsystems. Java 2 SDK, Standard Edition Documentation. `http://www.javasoft.com/products/jdk/1.3/docs/index.html`, 2001.

[MIM$^+$00] Akito Monden, Hajimu Iida, Kenichi Matsumoto, Katsuro Inoue, and Koji Torii. A practical method for watermarking java programs. In *the 24^{th} Computer Software and Applications Conference*, 2000.

[Mon00] Akito Monden. Java watermarking tools. `http://tori.aist-nara.ac.jp/jmark/`, 2000.

[MvOV97] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[NF98] David Nagy-Farkas. The easter egg archive. `http://www.eeggs.com/`, 1998.

[PA99] Fabien Petitcolas and Ross Anderson. Information hiding, an annotated bibliography, 1999.

[PAK98] Fabien Petitcolas, Ross Anderson, and Markus Kuhn. Attacks on copyright marking systems. *Second Workshop on Information Hiding*, pages 218–238, 1998.

[Par72] D.L. Parnas. On the criteria to be used in decomposing system into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PKK$^+$00a] Jens Palsberg, Sowmya Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Yi Zhang. Experience with software watermarking. In *Proceedings of 16th Annual Computer Security Applications Conference (ACSAC)*, New Orleans, Lousiana, December 2000.

[PKK$^+$00b] Jens Palsberg, Sowmya Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Yi Zhang. JavaWiz watermarking system. `http://www.cs.purdue.edu/homes/madi/wm/`, 2000.

[Sam94] Peter Samson. Apparatus and method for serializing and validating copies of computer software. US Patent 5,287,408, Assignee: Autodesk, Inc, February 1994.

[SHKQ99] Julien P. Stern, Gael Hachez, Francois Koeune, and Jean-Jacques Quisquater. Robust object watermarking: Application to code. In *Information Hiding*, pages 368–378, 1999.

[ST98] Tomas Sander and Christian Tschudin. On sofware protection via function hiding. In *2nd International Workshop on Information Hiding*, 1998.

[vV96] Hanpeter van Vleit. Mocha, the Java Decompiler. `http://www.brouhaha.com/∼eric/computers/mocha.html`, 1996.

[VVS01] Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *4th International Information Hiding Workshop*, Pittsburgh, PA, April 2001.

[Way96] Peter Wayner. *Disappearing Cryptography: Being and Nothingness on the Net*. Academic Press, New York, New York, 1996.