Tamper-proofing Software Watermarks

Clark Thomborson **Jasvir Nagra** **Ram Somaraju** **Charles He**

Computer Science Department
University of Auckland
New Zealand
Email: cthombor,jas,rsom012,yhe007@cs.auckland.ac.nz

Abstract

We introduce a novel method called *constant encoding*, which can be used to tamper-proof a software watermark that is embedded in the dynamic data structures of a program. Our novel tamperproofing method is based on transforming numeric or non-numeric constant values in the text of the watermarked program into function calls whose value depends on the watermark data structure. Under reasonable assumptions about the knowledge and resources of an attacker, we argue that no attacker can be certain that they have altered our tamperproofed watermark unless they take a risk of affecting program correctness in some way that may be difficult to detect. In this paper we also present a novel scheme for representing a numeric value as a Planted Plane Cubic Tree, and we describe how to use this scheme in a particularly-effective implementation of our constant encoding tamperproofing method.

1 Introduction

Anyone who tries to sell intellectual property in digital format will know (or quickly learn) of the existence of *pirates* who make unauthorised copies of digital objects for personal use or on-sale. There are several lines of defense against such pirates, for example technological measures to prevent copying or unauthorised use; legal measures (such as the Digital Millenium Copyright Act of the USA) to prevent copying, unauthorised use, or on-sale; and social measures (such as advertising campaigns) to decrease the motivation for piratical acts.

A related problem for a software producer is that of *reverse engineering*, which is the process by which a competitor may discover how to make a related software product even though the design documentation is not publically available. In an extreme case, the reverse engineer may take a copy of entire subroutines or even an entire executable, incorporating these subroutines or executable into a competing product. In many jurisdictions, such copying would be legally forbidden as a violation of the software producer's copyright; however it can be difficult for the vendor to discover that this violation has occurred.

In this paper we focus our attention on the technology of robust invisible software watermarks. The *robust* property ensures that such watermarks are difficult for a pirate or reverse engineer to remove. The *invisible* property indicates that these watermarks are not designed to be apparent to the end-user. Instead, the information in the watermark is intended to be revealed only to a digital-rights management (DRM) system that prevents or dissuades unauthorised use. In some cases, the DRM system must operate autonomously, for example if its intent is to prevent unauthorised copying or use on a desktop PC. In other cases, the DRM system is highly mediated by humans, for example when the evidence from a watermark detector is presented by an expert witness in courtroom proceedings where a copyright violation has been alleged.

Our preferred name for robust invisible watermarks is Prevention Marks, to emphasise their role in preventing unauthorised uses. There are three other major categories of watermarks. Any robust visible watermark is an Assertion Mark, used to make a public claim to ownership or some other public assertion. The results in this paper have some applicability to Assertion Marks.

Fragile invisible marks, or Permission Marks, should become illegible or invalid whenever the underlying object is changed or copied. For example, a DRM could use a Permission Mark to keep track of the number of authorised copies that can be made of a controlled object. The fourth and final class is the Affirmation Mark, in which a visible fragile watermark serves as a seal of authenticity. In prior writing (Nagra, Thomborson & Collberg 2002), we identified these functional categories but gave them different names. The tamperproofing techniques presented in this paper will increase the robustness of watermarks; hence they have little or no applicability to Permission Marks or Affirmation Marks.

1.1 Software Watermarking.

The process of software watermarking may be formally described as follows. Let $\mathbb{P}$ be the set of all legal programs in Java or some other fixed representation. We embed a robust watermark w into a program $P \in \mathbb{P}$, using embedder process $\mathcal{E}$ to produce program $P_w \in \mathbb{P}$ such that w can be reliably located and extracted from P_w, using extractor $\mathcal{X}$ even after P_w has been subjected to code transformations such as obfuscation, translation and optimization (Collberg & Thomborson 1999). In this paper we add a requirement of tamperproofing, so that the tamperproof robust watermark will also survive purposeful attacks by a skilled reverse engineer.

Ideally, a Prevention Mark w should have a mathematical property that allows us to argue that w did not arise by chance, but instead was embedded deliberately in the program. As an example of such a property, w may be an integer $x = pq$ that is a product of two very large primes p and q. The person who created this x can use standard cryptographic techniques (RSA Laboratories 2002) to demonstrate

their knowledge of its factors, whereas attackers will be unable to supply this proof of ownership.

If the mark w lacks a mathematical property it may be escrowed, and fully revealed only during courtroom proceedings where the authorship of the mark is contested.

Some watermarks are of the "Easter Egg" variety: they are only revealed when a specific input is presented to the program. To allow us to discuss such cases formally, we define $\mathbb{I}$ as the set of all the input sequences for programs in $\mathbb{P}$. For example in Java the input sequence may include key strokes, mouse strikes and input from other inputs defined in Java and its libraries. We will extend this formalism in Section 2. In the remainder of this section we briefly survey technologies for software watermarking.

1.2 Static and Dynamic Watermarks.

Software watermarks may be embedded either in the static representation of the program, or in its dynamic execution state. Static watermarks can be detected without running the program. This generally leads to an inexpensive detection process. However static watermarks are susceptible to attack by anyone of reasonable skill in software analysis who gains access to the detection system – this is typically a software program. Such an attacker may be expected to analyse the watermark detector to discover its operating principles. Once these principles are known, the attacker will generally be able to tamper with the watermark, either by rendering it unreadable or by modifying it to any desired value.

Static watermarks may be embedded in an otherwise-unused data area of a program. Such marks are trivially modified by anyone who knows how the embedder searches for the mark. A checksum could be included in the watermark, as noted in claim 8 of Holmes'407 (Holmes 1994), and such appendage would provide a modicum of tamperproofing. A successful attack on this tamperproofed watermark would require two steps: modifying the watermark, and modifying the checksum. However the second step is hardly more complex than the first, and could be trivially accomplished by any attacker who knows (or can discover) the checksum algorithm. We note in passing that a cryptographically secure checksum algorithm could be employed in Holmes' invention, however then the secret key for the checksum must be hidden somewhere. The search for this key (or for a working implementation of the keyed checksum-calculating algorithm) becomes a third step for our presumed attacker. If the watermark detector is distributed to end-users, for example as part of a license control system, then a skilled adversary will generally be able to access this extraction software for inspection, analysis, and modification of its checksum calculation routines.

Moskowitz and Cooperman (Moskowitz & Cooperman 1998) have patented a more advanced tamperproofing method for static software watermarks. In their technique, some portion of the program code is encoded into a watermark that is stored in some essential data area of the program, for example a digital image. The watermark also contains licensing information. Any attacker who makes random modifications to the data that obliterate or modify the licensing information, will almost certainly make random modifications to the portion of the program code that is encoded in the watermark. This will cause the attack to fail if the encoded code portion is carefully chosen to be essential to normal operation of the program. However, this method of tamperproofing will *not* be resilient to an attack by an adversary who is capable of analysing the tamperproofed code sufficiently to discover the encoded portions of the code. For example, an adversary may attach a debugger to the tamperproofed code. With a debugger, an adversary may generally observe program behaviour and outputs, whenever it makes reference to specified regions in its static data areas. An adversary may also observe any call to code that is constructed on-the-fly by a decoding process; once such a call is detected, the adversary may then modify the tamperproofed program so that its code region contains a static (unencoded) version of the formerly encoded portion. Then the adversary may freely distort the image containing the watermark until the watermark is unreadable, without fear of introducing "bugs" in program operation.

1.3 Dynamic Watermarks.

Collberg and Thomborson proposed a method of watermarking software called *dynamic data structure watermarks* (Collberg & Thomborson 1999). Such watermarked programs build a special, recognisable, data structure representing a particular graph w that serves as a watermark. This watermark may be of the Easter Egg variety, that is, it may be revealed by a special input sequence k.

In this paper we will make many references to the set of possible watermark graphs $\mathbb{G}$ available to a dynamic data structure watermarking scheme. The watermark w must be chosen from this set. As noted by Collberg and Thomborson (Collberg & Thomborson 1999), a good choice for $\mathbb{G}$ is the set of planted planar cubic trees of a given size, say those with 1000 leaf nodes.

A planted planar cubic tree (*PPCT tree*) has the following properties:

1. The tree is embedded in the plane.
2. All vertices are either monovalent or trivalent.
3. A single vertex is distinguished as the root of the tree.
4. The root is monovalent.

Whatever the specific choice of $\mathbb{G}$, a fundamental requirement on this set of graphs is that it be efficiently enumerable, in the following sense: $\mathbb{G}$ must have a associated pair of encoding and decoding functions (e, d), where the encoder function e of the codec maps a set of integers $\mathbb{N}_{|\mathbb{G}|}$ onto the set $\mathbb{G}$. The decoder d is the inverse of e, mapping elements of $\mathbb{G}$ onto the the set $\mathbb{N}_{|\mathbb{G}|}$. Here $\mathbb{N}_n$ denotes the set $[0, n)$ and $n \in \mathbb{N}$ is an integer. The pair (e, d) is referred to as a "graph codec".

The watermark embedder $\mathcal{E}$ uses the encoder function e to compute the graph $w = e(x)$ encoding some desired integer x, such as the $x = pq$ example discussed above. A data-structure representation $r(w)$ of w is embedded by $\mathcal{E}$ into the program P. This data-structure $r(w)$ might be created fully only when a specific keyed input $k \in \mathbb{I}$ is presented to the program P.

We now have developed enough formalism to describe the main features of our new technology and its predecessors. In Section 2 of this paper, we will complete our formalisation of the watermarking processes of embedding and extraction.

In our *constant encoding* method for tamperproofing watermarks, described fully in Section 3, we introduce randomly-chosen dependencies between the semantically-important operations of the watermarked program and the representation of the watermark graph in the dynamic data structures built by

the program at runtime. Specifically, the introduced dependencies are on the values of constants which, in the original (untamperproofed) program are merely loaded as integers (or other base type) from some statically-defined area (such as the constant pool in Java, or the constant-fields of individual instructions). In the tamperproofed program, these constants are decoded from the dynamic data structures of the program. The areas being decoded into constants may be part of the watermark w itself, in which case we have introduced a true dependency on the watermarked data structure $r(w)$. We also introduce false dependencies which are difficult for the attacker to distinguish from true dependencies, because the data structures r' which are decoded into constants will closely resemble the watermark $r(w)$, or because these dependencies are embedded in "dead code" guarded by opaquely false predicates.

Palsberg (Palsberg, Krishnaswami, Kwon, Ma, Shao & Zhang 2001) describes a somewhat related tamperproofing method for dynamic data structure watermarks. We discuss this method at the end of Section 3.

In Section 4 we argue that reasonably-limited attackers can only remove our tamperproof watermarks if they risk making undiagnosable changes to program correctness – unless the code being watermarked is so simple as to be completely testable by the attackers. However code that is simple enough to be completely testable by the attackers could, presumably, be completely redeveloped in a "clean room" by these attackers at reasonable cost. We note that watermark protection would be of little practical importance in such a case, and indeed it is unrealistic to expect to be able to embed a tamperproof watermark on software with a simple (and public, or easily discoverable) I/O behaviour.

In Section 5 we conclude our paper with some suggestions for future developments and enhancements.

2 Notation

As noted in the previous section, $\mathbb{P}$ is the set of all legal programs in Java or some other fixed representation, and $\mathbb{I}$ is the set of all the input sequences for programs in $\mathbb{P}$. In a dynamic data structure watermarking scheme, the watermark w must be chosen from a set $\mathbb{G}$ of graphs. A pair of functions (e, d) is referred to as a "graph codec", where the encoder e maps integers $\mathbb{N}_{|\mathbb{G}|}$ onto the set $\mathbb{G}$, and decoder d is the inverse of e. Here $\mathbb{N}_n$ denotes the set $[0, n)$ and $n \in \mathbb{N}$ is an integer.

The watermark embedder $\mathcal{E}$ uses the encoder function e to compute the graph $w = e(x)$ which corresponds to some desired watermark integer x. This graph is then embedded into the program P. More precisely, what is embedded in the program is some representation $r(w)$ of the graphical watermark w, where r is a function $r : \mathbb{G} \to \mathbb{S}$ that maps graphs in set $\mathbb{G}$ onto the set $\mathbb{S}$ of data structures that may be used by programs in $\mathbb{P}$. This mapping and its inverse $r^{-1} : \mathbb{S} \to \mathbb{G}$ must be efficiently computable.

A suitable representation of graphs as members of $\mathbb{S}$ makes use of the pointer-type data structures that are commonly encountered in computer programs written in Java, C or C++. In numerical programs, one could represent a graph by an array of integers, where the rows of the array represent the nodes of the graph, and the columns of the array represent the arcs of the graph. Many other representations could be quickly devised and described by anyone conversant with graph algorithms.

We require, of our graph representations, that they specify some total order on the outgoing arcs at each node. That is, there must be a "first" (according to the total order) outgoing arc at each node that has at least one out-arc. This constraint is easily satisfied by most common methods for representing graphs in data structures. For example, in the pointer-type data structure representation, the total order on the outgoing arcs is defined by the order in which the corresponding pointer references appear in computer memory.

From this point forward in this paper, the term graph refers to a directed graph, possibly disconnected, with a total ordering on the outgoing arcs at each node.

We can now define the embedding function $\mathcal{E}$ used in a data structure watermarking algorithm as follows:

$$\mathcal{E} : \mathbb{P} \times \mathbb{N}_{|\mathbb{G}|} \times \mathbb{I} \to \mathbb{P} \tag{1}$$

This embedding function $\mathcal{E}(P, w, k)$ takes a program P, a watermark integer $w \in \mathbb{N}_{|\mathbb{G}|}$ and a secret key input sequence k and produces a Program P_w, such that, when P_w is run on the key input k, it produces a data structure S_w such that $d(r^{-1}(S_w)) = w$.

The corresponding extractor function

$$\mathcal{X} : \mathbb{P} \times \mathbb{I} \to \mathbb{G} \tag{2}$$

takes a watermarked program P_w, runs it on a given input $i \in \mathbb{I}$ and "de-represents" the data structure S_w built by P_w, returning the graph $r^{-1}(S_w)$. When the extractor is run on P_w and the special input k, the watermark is revealed: $d(\mathcal{X}(P_w, k)) = w$.

A degree of invisibility can be imposed by requiring that, for at least some $k' \neq k$, $d(\mathcal{X}(P_w, k')) \neq w$. In some applications such extreme invisibility is irrelevant or unimportant, indeed it may be desirable to build the watermark w before any input is received, in which case the special input k is the (easily guessed) null input sequence Φ.

An extreme degree of invisibility can be imposed by using a watermark that has a Boolean-valued recognition function $\mathcal{X}'$ but no efficient extractor function $\mathcal{X}$. Such recognition functions $\mathcal{X}'(P, k', w')$ return the value *true* (signifying recognition) only when the watermark specified in their third argument w' matches the watermark found when program P is run on input k'. Watermarks lacking efficient extractor functions are sometimes called *private watermarks*. They generally suffer from the problem that their robustness depends on the recognition value w being a closely-held secret. For example, DRM systems based on private watermarks should not be released into any environment where an attacker might discover the secret w by reverse-engineering, and then use this knowledge (in a *subtractive attack*) to create a pirated derivative product that does not bear the watermark w.

The primary constraints on any tamperproofing process are that it must not be too expensive either in time or in space; and the recognition properties (cost of recognition, visibility/invisibility, fragility/robustness) of the watermark must not be affected by tamperproofing. The goal of tamperproofing a watermark is to make it economically infeasible for a skilled reverse engineer to remove or modify the watermark, where such removal or modification must be guaranteed to preserve program correctness.

3 Tamperproofing

The main idea behind constant encoding is to replace constants used in programs with a function f whose value is dependent on the values of pointer variables

in the dynamic data structure S that contains the watermark. We take special care to ensure that any portion of S that will affect the value of our f, will be properly initialised before any call is made to f.

We implement our constant encoding method with an algorithm $\mathcal{T} : \mathbb{P} \times \mathbb{Z} \rightarrow \mathbb{P}$ having the following properties. Implementation notes on this algorithm are presented later in this section.

1. The inputs to $\mathcal{T}$ are a watermarked program P_w and an integer c.
2. The output of $\mathcal{T}$ is a modified program P'_w with the same watermarking behavior: for all inputs $i \in \mathbb{I}$
$$d(\mathcal{X}(P'_w, i)) = d(\mathcal{X}(P_w, i)) \quad (3)$$
3. *Semantic Equivalence.* The observable behavior of P'_w does not differ from P_w in any important manner, that is, $\mathcal{T}$ preserves the semantics of the program P_w without greatly changing its runtime or consumption of system resources.
4. The algorithm $\mathcal{T}$ may be executed repeatedly, until a desired amount of tamperproofing is acheived.
5. The algorithm $\mathcal{T}$ selects a statement p_c in P_w, where p_c is chosen uniformly at random from the set of all statements using as c a constant.
6. The program P'_w constructed by $\mathcal{T}$ differs from P_w at the statement p_c, the constant-loading portion of which is replaced by a function call $f(a_1, a_2, ..., a_n)$.
7. The arguments $(a_1, a_2, ..., a_n)$ and the function f are chosen appropriately by algorithm $\mathcal{T}$, to ensure that the result value of the function call $f(a_1, a_2, ..., a_n)$ is invariant over all execution paths leading to this call. This guarantees the semantic equivalence of P'_w and P_w, as required by property (3) above.
8. *Dependency property.* One or more of the arguments $(a_1, a_2, ..., a_n)$ should be pointers into the data structure S built by $P'w$. These arguments may reference areas of the data structure in which the watermark is embedded. The desired property of the pointer arguments is to provide tamperproofing of the watermark, in the following sense: if the data structure is altered indiscriminately by an attacker, in an attempt to remove its watermark, then $f()$ may evaluate incorrectly and the program semantics may change.
9. *Many-to-one property.* Randomisation (or other means unpredictable to the potential attacker) should be used in the selection of f, and of the value of each its arguments $(a_1, a_2, ..., a_n)$. This random selection should be made over an extremely large range of possible variants that would evaluate to the same constant c; in mathematical terminology, the function f should be many-to-one. The desired property of this randomised selection process is to prevent any reverse engineer from building a small but comprehensive catalog, or other compact description, of all $f(a_1, a_2, ..., a_n)$ that may appear in a tamperproofed program.
10. One or more of the arguments $(a_1, a_2, ..., a_n)$ may be integer constants.
11. *Resistance to protocol attack.* The decoding function d used by the extractor function $\mathcal{X}$ should be functionally present in the tamperproof watermarked program, so that the watermark is resistant to the protocol attack analyzed in Section 4.6.
12. *Stealth and invariance properties of watermarked data structures and decoys.* The algorithm $\mathcal{T}$ may, in an initial step, modify the program P_w so that the modified program P'_w builds a modified data structure S' with the desirable properties of *stealth* and *invariance*, described briefly below. The data types and operations used to create new regions of, or to modify existing regions of, the data structure S of the original program P_w should closely resemble the data types and operations used to create the watermarked data structure(s) in P_w. This stealthiness or close resemblance will make it difficult for an attacker to distinguish the modified regions from the watermarked regions. The desired invariance property of S' is that algorithm $\mathcal{T}$, in any of its (possibly repeated) applications, will have efficient means of discovering (or recalling, by table lookup or other means of memorisation) pointers or references into S' that have desirable values as arguments of f, where these desirable values are invariant over all execution paths leading to function call f.
13. *Invariance property of f.* The function f should have a desirable invariance property such that algorithm $\mathcal{T}$, in any of its (possibly repeated) applications, will have efficient means of discovering (or recalling, by table lookup or other means of memorisation) pointers or references into S' whose variation, over all execution paths leading to function call f, can not affect the value of f. For example a function f would have the desirable invariance property if its value were unaffected by the structure of the right-child descendants (if any) of its first argument, where this first argument is a reference to a representation of node in a binary tree. This would be a desirable function for the tamperproofing of a region of data structure S' representing a node of a watermark tree with a known (invariant) structure in its left-child.
14. *Variable dependency.* The algorithm $\mathcal{T}$ should have the capacity to modify the program P_w so that the modified program P'_w has a program variable whose presence is necessary for correct operation, with the property that the current value of this variable is decoded by a function call f of the form described above. Alternatively, this modification to P_w may be done by a human operator of $\mathcal{T}$. The desired property of this introduction of variable dependency is to prevent an attacker from mounting a possible "pattern-matching" attack on the tamperproof watermarked program, as discussed in Section 4. As an example of an introduced instance, a program loop may be unrolled once, allowing a variation in program coding such that even-numbered iterations of the loop require a "True" value of a newly-introduced Boolean variable for correctness, while odd-numbered iterations of the loop require a "False" value for correctness.
15. *Dead code property.* The algorithm $\mathcal{T}$ should have the capacity to modify the program P_w so that the modified program P'_w has function calls f of the form described above in "dead code"

that will never be executed. This is another defense against a pattern-matching attack.

3.1 Example

Consider the following trivial program that builds a watermark w encoding the value 7.

```
public class A {
    PPCT w; // w is our watermark
    public void print( ) {
        w = build_PPCT_watermark( 7 );
        int a = 2;
        System.out.println( a );
    }
    public static void main( String[] args ) {
        new A( ).print( );
    }
}
```

If the constant "2" were chosen for tamperproofing, this trivial program might be revised as follows.

```
public class A{
    PPCT w;
    PPCT a1, a2, a3; // pointers into w
    public void print( ){
        PPCT s; // a subtree of g
        w = build_PPCT_watermark(7, a1, a2, a3);
        // a1, a2, a3 are pointers into w
        s = t(a1, a2, a3); // subgraph of w
        a = d(s);  // decode 2 from s
        System.out.println(a);
    }
    public static void main( String[] args ){
        new A( ).print( );
    }
}
```

The revised program uses pointers $a1$, $a2$, and $a3$ into the watermark graph w as arguments to a function t whose implementation will be discussed later in this section. The value of $t(a1, a2, a3)$ is a PPCT s with the property that $d(s) = 2$, where d is the decoding function of the watermark codec. In terms of the generalised properties listed above for the tamperproofing algorithm, we have implemented the function $f(a_1, a_2, ..., a_n)$ as the composition of the decoder d and the PPCT-valued function t.

3.2 Implementation Notes

The first step in the implementation is to perform an interprocedural flow analysis on P_w, to discover one or more code segments ("dominators") that are guaranteed to execute prior to the execution of the constant-loading statement p_c. The next step is to insert new program statements into one or more of these dominating code segments, where these new program statements have the effect of allocating new objects in the program's dynamic data structure, and setting the pointer fields a_i of these objects so that they represent any desired graph as required to get the appropriate constant value c as the result of evaluating $d(t(a_1, a_2, ..., a_n))$. Some of these pointer fields may refer to components of the watermark w that are known to be constant over the entire program run. A full set of such invariants on w would be difficult to discover by any static analysis of P_w, however these invariants could be recorded during the watermarking process and transmitted as ancillary information to the tamperproofing process.

For example, the watermark w may be in the shape of a tree, in which case the ancillary information may report that the subtree rooted at some node x will remain unchanged after the first execution of program statement y. In this case the node x, or any of its descendants, may be used as a constant-valued argument for an introduced function f that replaces the constant-loading portion of any program statement dominated by statement y.

The selection of an appropriate function f, and of an appropriate set of arguments $(a_1, a_2, ..., a_n)$, may be deferred until the runtime of the tamperproofed program P'_w. For example the constant-loading part of a single program statement p_c in the watermarked program P_w may be replaced by a conditional program statement, in which the second alternative is executed in cases where an execution of the first alternative would not yield in a calculation of the desired constant result c. A convenient "switch" to control such case statements could be obtained by adapting the path profiling algorithm of Ball and Larus (Ball & Larus 1996). This, or some similar technique in which a program may gather information about its own path of execution, would allow the newly introduced program variable(s) to control the evaluation of f as well as to invoke any required initializations of the data structure(s) referenced by its arguments.

The preservation of the watermark, and of all observable behaviour of P_w, can be guaranteed by using only semantics-preserving program transformations during the tamperproofing process, along with a normal level of care for program efficiency so that the tamperproofing does not result in an easily-observable slowdown of program operations. A normal level of care for program efficiency would generally include the following measures if a program flow analysis reveals that p_c may be executed multiple times.

1. The evaluation of f, in code replacing the constant-loading portion of p_c, may be hoisted to some dominating program segment that is executed at most once.

2. The evaluation of f, in code replacing the constant-loading portion of p_c, may be guarded by a predicate (initially true) dependent on one or more newly introduced program variables. The value of one or more of these newly introduced program variables may be adjusted during the first execution of the code replacing p_c, so that any subsequent executions of the code replacing p_c may reuse the result of an earlier evaluation of f. Occasional re-evaluations of f may occur (instead of a reuse of a stored constant value) without noticeable impact on program performance.

3.3 Possible inclusion of non-constant function evaluations f

Any program variable whose value is rarely updated may be treated as a constant during the periods in which it is unchanged. All evaluations of such a non-constant variable may be replaced by an evaluation of a function f as described above, if the data structure referenced by the arguments of this function is modified every time the non-constant variable is updated. For example, an evaluation p_v of a variable whose value is initially false, and whose value is set to true after some point y in program execution, may be replaced by a function evaluation on one or more non-constant arguments whose values are varied by newly-introduced program statements at point y in the tamperproofed program.

3.4 Avoiding a Chicken-and-Egg Conundrum

As noted in the introduction, our tamperproofing method bears some similarity to a method of obfuscating and tamperproofing a watermark, which was proposed several years ago by Palsberg et al. (Palsberg et al. 2001). In Palsberg's method, a graph w' is chosen from the same set $\mathbb{G}$ as the watermark w. A data structure representing the graph w'

is built at the very beginning of the execution of the watermarked program; the code that builds this data structure is added by Palsberg's obfuscating and tamperproofing process. This data structure w' is analogous to our modified program data structure S'.

Palsberg's method also inserts opaque predicates (Collberg, Thomborson & Low 1998) of the form `x==y` or `x!=y`, where `x` and `y` are pointers into w'. Opaque predicates that evaluate to the constant value `true` are used to guard semantically-important regions of the watermarked code; and opaque predicates that evaluate to the constant value `false` are used to guard spurious code, inserted during Palsberg's method, that if executed would damage the correctness of the watermarked code.

We believe that Palsberg's tamperproofing is effective against transformative attacks by a limited adversary, however an expert adversary who is able to distinguish w from w' may be able to (in Palsberg's words) "unravel the whole construction."

Our constant-encoding method can be seen as a greatly extended variant of Palsberg's chicken-and-egg method. Palsberg's method encodes only the logical constants `true` and `false`, and this encoding uses only a simple function f that consists of an equality test on two pointer variables. Our method encodes any integer value, using much more complex functions f of many variables.

Our method is sharply distinguished from Palsberg's method, in that we are replacing constants that occur in the watermarked code. By contrast, Palsberg introduces Boolean constants as guards (opaque predicates) to the watermarked code.

Finally, and perhaps most importantly, our method is sharply distinguished from Palsberg's method because (in one of the alternatives described above) the value of our constant-encoding function f may depend on portions of the watermark w tree that are built unconditionally by code segments that dominate the constant-loading statement p_c. This is a solution to Palsberg's chicken-and-egg conundrum, for in our method a (constant portion of) a watermark w may be used to defend the code that modifies or builds other portions of w.

3.5 Tree-Valued Functions

Our simplest tree-valued function has a single argument: $t(a_i)$ is the set of all data structure nodes reachable from a_i with a depth-first search. Note that we use the total ordering on the outgoing arcs from each node to unambiguously define the depth first search. Also, note that $t(a_i)$ is a sub-graph of the entire graph represented by the data structure S built by P_w.

We define the intersection of two trees, $t_1 \wedge t_2$, in the natural way. If either tree is null then the intersection is null. In general, if the root of t_1 has j_1 children and the root of t_2 has j_2 children, then the root of $t_1 \wedge t_2$ has $\min(j_1, j_2)$ children. The structure of the subtrees rooted at each of these children is defined analogously, in a recursive fashion; for example if $\min(j_1, j_2) > 1$ then the leftmost child of $t_1 \wedge t_2$ has $\min(j_{11}, j_{21})$ children if the leftmost child of t_1 has j_{11} children and the leftmost child of t_2 has j_{21} children.

These ideas are illustrated in Figure 1. Part (a) of this figure shows a graph whose nodes are labelled by a depth-first search beginning at the node labeled 0 referenced by pointer a_3. Part (b) of this figure shows the tree $t(a_1)$. Part (c) shows the tree $t(a_2)$, and part (c) shows the intersection $t_1 \wedge t_2$.

3.5.1 Masking Functions

Our simplest two-argument tree-valued function is the "masking function" $t_m(a_1, a_2)$ defined as the intersection of $t(a_1)$ and $t(a_2)$. We call $t_m()$ a masking function because the tree represented by its second argument is used to "mask" (or filter) the nodes in the tree represented by the first argument. We note that generally t has the desired many-to-one property for a large tree, which typically has many subtrees whose intersection is a desired tree such as the one in Figure 1. For example in this figure, $t_m(a_1, a_2) = t_m(a_1, a_3)$.

Note that the value of a masking function is generally insensitive to small variations in one or more of its arguments. For example the value of $t_m(a, b)$ is unaffected by any changes to the right-hand subtree (if any) of its second argument b if its first argument a has no right-hand subtree. Thus this function, with argument a known to have no right-hand subtree, would be suitable for the tamperproofing of a region of data structure S' representing a node b of a watermark tree with an invariant left subtree and a variable right subtree.

Extension 1. Any argument a_i in a masking function may be replaced by an integer encoding a binary tree as a totally balanced sequence (see Extension 4 below). Thus many different argument lists $(a_1, a_2, ..., a_n)$ will generate the same constant c, increasing the difficulty of pattern-matching attacks.

Extension 2. We define the union of two trees analogous to the intersection operation, but replacing min() by max() in the recursive definition. A masking function may then be any tree valued function obtained by union and intersection. For example we could define a three argument t_{m3} as

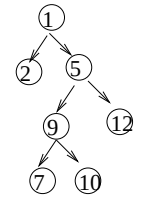
$$t_{m3}(a_1, a_2, a_3) \doteq (t(a_1) \wedge t(a_2)) \vee t(a_3) \qquad (4)$$

3.5.2 Boundary Functions

Our boundary function $t_b(r, a_1, a_2, ..., a_n)$ has a first argument r defining a sub-tree $t(r)$ using a depth-first search. The remaining arguments $(a_1, a_2, ..., a_n)$ define nodes acting as "Boundaries" that cut off portions of $t(r)$ by the following algorithm.

1. Perform a depth first search to discover the nodes of $t(r)$, terminating the search whenever any node in $(a_1, a_2, ..., a_n)$ is encountered.
2. Return a tree composed of all nodes encountered in the search, excluding the terminating nodes.

See Figure 2 for an example, showing $t_b(a_1, 3, 4, 6, 11)$.



The tree $t(a_1, 3, 4, 6, 11)$

Figure 2: The tree $t(a_1, 3, 4, 6, 11)$ derived from the data structure of Figure 1a.

Note that the list $(a_1, a_2, ..., a_n)$ may contain nodes that are not found in the search of $t(r)$ and hence t_b is a many-to-one function. For example the tree in Figure 2 can be encoded as $t_b(a_1, 3, 4, 6, 11, 8)$. The desired invariance property is also present in t_b because one or more of the boundaries may be arbitrary references to watermarked portions of the data

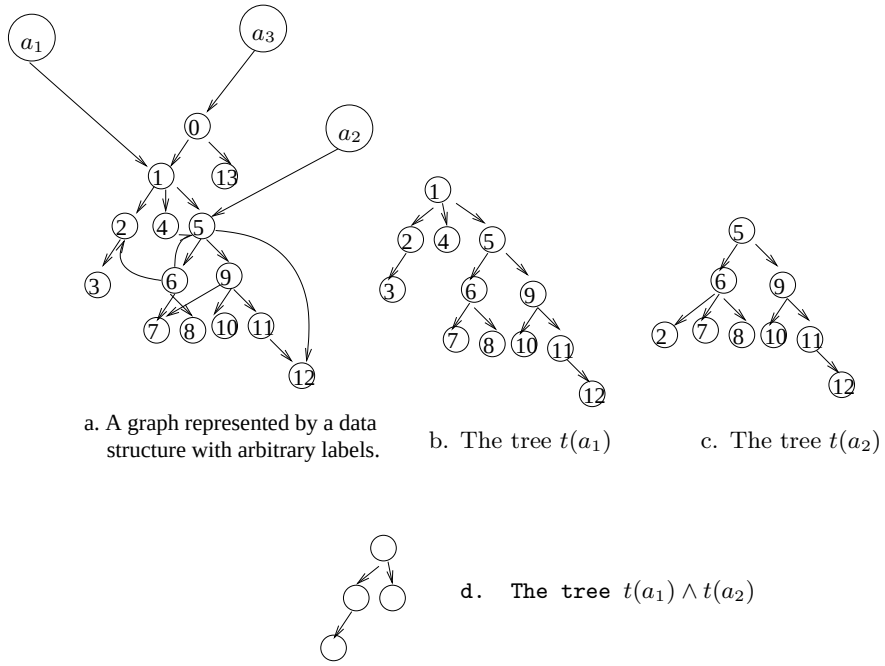


Figure 1: Illustrating depth first search trees and their intersection.

structure, in any context where algorithm $\mathcal{T}$ has determined that subtree $t(r)$ is disjoint from the watermarked portions of the data structure.

3.6 Encoding and Decoding Functions

We use well-known techniques from combinatorial graph theory to design codecs that convert integers into trees and vice versa. Two implementations of these techniques will be described very briefly below. Both implementations are based on PPCT trees.

It is easily shown (Kreher & Stinson 1999, Palsberg et al. 2001) that PPCT graphs are enumerated by the Catalan numbers C_n where

$$C_n = \frac{\binom{2n-2}{n-1}}{n} \tag{5}$$

We define the set of all PPCTs with n leaves to be $\mathbb{G}_n$. Where the context is clear we suppress the subscript n and write $\mathbb{G}$ instead of $\mathbb{G}_n$. Also let $\overline{\mathbb{G}}_n = \mathbb{G}_1 \cup \mathbb{G}_2 \cup ... \cup \mathbb{G}_n$.

3.6.1 Codec 1

The codec (d_1, e_1) was implemented by Yong He (He 2002), who corrected an erroneous term in the recursive formulas of Palsberg et al. (Palsberg et al. 2001). It ranks left-balanced trees higher than right-balanced trees. The decoder $d_1 : \mathbb{G}_n \rightarrow \mathbb{N}_{|\mathbb{G}_n|}$ is defined recursively as follows:

$$d_1(\mathcal{G}) = \begin{cases} 0 & \text{if } |\mathcal{G}| = 1 \\ d_1(\mathcal{G}.\mathcal{L}) * C_R + d_1(\mathcal{G}.\mathcal{R}) & \\ \quad + \sum_{i=1}^{L-1} C_{L-i} * C_{R+i} & \text{if } |\mathcal{G}| > 1 \end{cases} \tag{6}$$

Here $\mathcal{G}.\mathcal{L}$ and $\mathcal{G}.\mathcal{R}$ are the left and right sub-trees of $\mathcal{G}$, with $L = |\mathcal{G}.\mathcal{L}|$ and $R = |\mathcal{G}.\mathcal{R}|$.

Extension 3. In this extension, the decoder function d_1' is defined as $d_1' : \overline{\mathbb{G}} \rightarrow \mathbb{N}_{|\overline{\mathbb{G}}|}$. This definition allows the PPCTs to have a variable number of leaves. Curiously, the same recursive algorithm is used for d_1' as for d_1, because there is no dependence on the number of leaves n in the recurrences presented above for d_1.

The recurrence relations above are rather time-consuming to calculate for trees with hundreds or thousands of leaves, because the arithmetic must be performed to hundreds or thousands of bits of precision. Almost two bits of precision are required for each leaf in a PPCT; for example, there are $C_{200} \approx 2^{388}$ PPCTs with 200 leaves, so 388 bits of arithmetic precision are required in the calculations made by any codec for such trees. The calculation $d_1(\mathcal{G}.\mathcal{L}) * C_R$ in the recurrence above requires one high-precision multiplication per node in the tree. Many additional high-precision multiplications are required to compute the summation in the third term for $d_1(\mathcal{G})$.

Extension 4. PPCT graphs, and all other Catalan-enumerable combinatorial objects, are in a 1-1 relationship with the "totally balanced binary sequences", for which a multiplication-free codec was developed by Kreher (Kreher & Stinson 1999). This codec builds up a two-dimensional table of dimension n by n, using a Pascal-triangle recurrence involving a single addition for each cell in the table, to encode or decode an n-element sequence. The functions $d_1()$ and $e_1()$ on n-leaf trees may be evaluated with $O(n)$ addition operations per function evaluation, after this table is precomputed in $O(n^2)$ addition operations.

It would be reasonably straightforward to adapt Kreher's codec to operate directly on PPCT graphs, although it seems somewhat easier to write linear-time translation algorithms to convert a PPCT graph into a totally balanced binary sequence and *vice versa*. Indeed, such translation algorithms were derived in Charles (Yong) He's recent Master's thesis on tamperproof software watermarks(He 2002), even though Charles was not aware of Kreher's multiplication-free codec.

3.6.2 Codec 2

We have done some experimentation with an alternative codec (d_2, e_2) in which the PPCTs are enumerated in order of increasing depth.

3.7 Choice of Arguments

We have conducted a series of Monte Carlo experiments and proven various lemmas, of the sort described briefly below, to verify that each integer constant commonly occurring in a computer program may be decoded (by the decoding functions described in the previous subsection) from an extremely wide range of arguments, for any of of the masking or boundary functions described earlier in this paper.

For example if we choose a random integer x, where x is uniformly distributed over the range $[0, C_{200} - 1]$, then the PPCT watermark $w = e_1(x)$ encoding this integer will have 201 leaves when Codec 1 is employed. If two nodes a and b are chosen uniformly at random from among the nodes of this randomly chosen watermark tree, then the integer decoded from the simplest 2-input masking function $t_m(a, b)$ will have a probability greater than 75% of being either a 0 or a 1. This is easily verified as a consequence of the following facts: there are $400^2 = 160000$ different ways of selecting two nodes a and b from a tree with 201 leaves; there are more than 120000 different ways to select two nodes a and b such that at least one of these two nodes is either a leaf or a node at distance one from a leaf; and $d_1(t_m(a, b))$ will be a 0-1 integer whenever $t_m(a, b)$ is a tree with one or two leaves.

Thus our simplest 2-input masking function strongly exhibits the desired "many-to-one" property for the commonly encountered constants zero and one. Furthermore, because a zero is always decoded by this function if one of its two arguments is a leaf node, this function has the desired invariance property.

Constants larger than "0" or "1" may be decoded from trees as well, even though our Monte Carlo experimentation shows that the many-to-one property of the simplest 2-input masking function falls off sharply with the size of the integer constant. For example, with probability in excess of 90%, all integers in the range 0 to 63 can be decoded by $d_1(t_m(a, b))$ for at least one selection of a and b, where arguments a and b are taken from the nodes of a randomly-chosen 201-leaf watermark tree w. Our alternative codec (d_2, e_2) gives somewhat higher probabilities, although the differences are generally not dramatic. Increasing the size of the randomly-chosen tree w can greatly increase the probabilities. Our results can be roughly characterised as indicating that a randomly chosen PPCT with n leaves can encode a randomly chosen integer in the range 0 to $n/3$, with probability in excess of 90%. Integers encoded as unbalanced trees are (unsurprisingly) less likely to be encodable than integers encoded as balanced trees, so our Codec 2 can handle a slightly larger range of integers than Codec 1, for a PPCT of fixed size.

To decode large constants, with the desired many-to-one property, the more complex masking and boundary functions described in this article may be employed. Alternatively, the well-known technique of bit-string concatenation may be employed, for example a 2-bit constant may be constructed by concatenating two 1-bit constants that are decoded individually from trees referenced by simple masking or boundary functions.

4 Security analysis

In order to facilitate a discussion of the strengths and weaknesses of our tamper-proofing technique, we introduce a more formal definition of an "attack".

Let $\mathbb{A} : \mathbb{P} \rightarrow \mathbb{P}$ be an "attack set": a collection of program transformations that an attacker has selected for possible use against a watermarked program. Any competent attacker will select a set containing only transforms that are extremely likely to preserve program correctness. This means the transformed program $a(P_w)$ should not differ in any obvious behavioural way from the original program P_w. The rational attacker will also attempt to select only transforms that are extremely likely to modify the watermark, that is, $d(\mathcal{X}(a(P_w), k)) \neq d(\mathcal{X}(P_w, k))$.

We say that an attack is successful, if it modifies the watermark and preserves program correctness.

The expert reader may note that our notion of an attack set $\mathbb{A}$ is general enough to encompass any semantics-preserving transformative attack, subtractive attack, or additive attack (Collberg & Thomborson 1999).

As noted in the previous section, users of our tamper-proofing method should take reasonable steps to prevent the attacker from learning our key input k, our extractor function $\mathcal{X}$, or our watermark integer w. This lack of knowledge will prevent the attacker from being able to determine, with high confidence, whether or not an attack was successful. An optimal strategy for such knowledge-limited attackers is to examine the watermarked program P_w to the best of their skill, within their time and resource constraints. The goal of this examination is to allow the attacker to construct an attack set $\mathbb{A}$ for P_w that maximizes the likelihood of a successful attack, if an attack is chosen probabilistically from this set. Note that an attacker who lacks knowledge of k may not be able to construct a set containing a single attack that will be successful against any k. Furthermore a probabilistic choice on the part of an attacker is an optimal strategy, according to an elementary game-theoretic analysis, against a defender who may choose a suitable watermarking scheme in partial knowledge of the attacker's eventual strategy. A similar analysis shows that the defender should make a probabilistic choice of their watermark, for any deterministic choice will have little secrecy against to a knowledgeable attacker.

We would expect any competent attacker to conduct a sequential attack, in which the attack sequence consists of the selection of any available version of the watermarked program, the definition (or redefinition) of an attack set appropriate for the selected version, the random selection of an attack from the set, the application of the selected attack thereby creating a new version of the watermarked program, an evaluation of the success of the current attack, and a decision on whether or not to continue the attack by repeating the sequence. The continuation decision will depend on the resources (time and computational budget) available to the attacker, and upon the attacker's estimate of the likelihood that they can correctly identify a dewatermarked version of the program.

4.1 Attacks

There are several techniques that an attacker may choose to use to remove a watermark embedded in a program. One class of techniques involves altering the embedded watermark graph itself. In order to achieve this it is necessary to:

1. Replace all calls to $f()$ by the constant value returned by $f()$.

2. Modify/remove the watermarked data structure S.

Without our tamper-proofing, an attacker who guessed or deduced the location of S and altered S indiscriminately, could do so without endangering program correctness. However an attacker who is faced with a tamper-proofed program must find and remove all functions $f()$ that depend on the dynamic data structure of the watermark; only then can the watermark safely be changed. In terms of our model, this means that a successful attack set can be constructed only after an attacker analyzes the program sufficiently deeply to (reasonably accurately) list all calls to $f()$. This analysis may be static, interpretative, or dynamic; it may also involve pattern matching.

4.2 Static Analysis

One possible attack on a tamper-proofed P'_w involves a static analysis of the program to find the result of a call to function $f()$. Such a "static analysis" attack is unlikely to be feasible, because the value returned by our function $f()$ depends on the values of pointers stored in dynamically-allocated objects, and such functions are very difficult to analyze statically (Collberg et al. 1998, Palsberg et al. 2001).

4.3 Dynamic Analysis

A more powerful method of attacking the program is available if the attacker is willing to execute a program. If the attacker is able to recognize our tamper-proofing function calls $f()$ in the tamper-proofed program P'_w, they may then observe the result computed by this call to $f()$ during a program run. We call this a "dynamic attack" because it presumes that the attacker is able and willing to execute and observe the tamper-proofed code in its dynamic environment. After successfully identifying the function return point, and after accurately observing the constant result, the attacker may replace the function call by a load of a constant. The attacker must iterate this attack as many times as we have iterated our tamper-proofing method $\mathcal{T}$, before all references to our modified data structure S' will be removed from the tamper-proofed code P'_w. After all these references are removed, it may be safe for the attacker to remove all code that references or allocates objects of the same data type as S'. This will successfully remove the watermark w without damaging the semantics of P_w, so long as the dynamic data structure watermark w is not built from objects of some data type that already existed in the unwatermarked program P.

Recall that our goal in tamper-proofing is to increase the attacker's uncertainty as to whether or not they have successfully removed the watermark.

An expert and determined attacker could succeed in a dynamic attack, if the watermark w and some constant structure w' in the modified data structure S' are the only structures that use a particular data type that is recognizable to the attacker. Such attacks could be frustrated by an additional process step, prior to the iterative application of our tamper-proofing method $\mathcal{T}$. Our recommended process step is the iterated use of an obfuscater to add many instances of spurious code, guarded by an opaquely-false predicate, before tamper-proofing. The dynamic attacker, even after running such code under many imaginable inputs and for a long time, will never gain complete confidence that their attack set is likely to remove all "live" references to objects of the data type used for w and w'. Here we are relying on the theoretically-sound notion of the undecidability of the halting problem, and the practically-unsolved problem of creating a covering set of test inputs for any reasonably-complex piece of real-world code.

4.4 Interpretative analysis

A closely-related possibility is an interpretive attack, in which attacker seeks to discover the result of a specific call to function $f()$ by interpreting the statements in $f()$ on a virtual machine. Here we are distinguishing an interpretation on a virtual machine from a dynamical execution of these same statements in an actual run-time environment of P_w. In general, interpretation takes much more time per statement than a dynamic execution; however an interpretive environment gives the attacker much more observational and analytic access to the execution state and history. The slower speed of the interpretation is a great liability to an attacker, whenever the tamper-proofer confuses the attacker (i.e. by opaquely-false predicates as noted in the previous paragraph) about the liveness of a code segment. We are not aware of any interpretive environment that would give much assistance to an attacker who wishes to visualize and analyze the graphs (such as planted plane cubic trees) that may be represented by a segment of a dynamic data structure. The effort required to construct and maintain such an environment is, we tentatively assert, a significant barrier to a successful interpretive attack on our tamper-proofed watermark. Our tamper-proofing is modestly potent even against such expert and well-resourced attackers, because our tamper-proofing introduces a data structure that closely resembles the watermark w.

4.5 Pattern-Matching Attacks

Any of the attacks listed above may involve some pattern matching on the static representation of the program, in which an attacker hypothesizes (and eventually discovers) a pattern or other distinctive signature of all function calls $f()$ inserted by the tamper-proofing.

Pattern-matching on the static representation of a program may be employed in conjunction with a dynamic or interpretative analysis, in which the attacker observes the operation of the program using a debugger or other means, to discover the value returned by every $f()$ that is recognised by the current pattern-matching hypothesis. Any hypothesized $f()$ which returns a non-constant value, over all observations, is evidence against the current hypothesis. The hypothesis may then be adjusted to be consistent with all observations to date. At any point, the attacker may choose to test the hypothesis by replacing all $f()$ with a best-guess constant value. If the modified program seems to work accurately, the attacker may subsequently modify the watermark without any immediately-obvious damage to program correctness.

To counter this pattern matching attack, the function f should be a many-to-one function, so that different occurrences of the same constant c may be replaced by calls to function f with different parameter lists. Another defense, noted in point 14 of Section 3, is the introduction of calls to $f()$ that produce non-constant values during program runs. For example, when a constant 0 is required in the program, this may be computed as the sum $f(y) + f'(y)$ where these two calls to $f()$ produce different values x and $-x$ on different program runs. Such non-constant $f()$ would complicate a pattern-matching attack, for an attacker must recognise a source code sequence computing $f(y)+f'(y)$, for any y, as a pattern that always produces a 0. This idea may be extended indefinitely,

in an "arms race" where the defender introduces patterns with $z > 1$ variables to counter an attacker who can recognize patterns with at most z variables. This arms race is moderated by the commonly-encountered tradeoff between security and performance.

If the runtime complexity of the constant-loading functions were doubled, for example by using $f(y) + f(y')$ instead of $f(y)$ to compute a constant, then only half as many of our watermark "guard" functions could be executed in any program run. At present, our technique is so novel that no attacker is likely to recognize a complex pattern. Thus we believe that early applications of our technique should guard watermarks with as many simple guards as possible without unacceptable performance degradation. More complex guards should be employed when pattern-matching attack tools have become easier and more attractive to build, that is, after many programs have been protected by our method.

4.6 Protocol Attacks

We close our security analysis by considering a "protocol attack" on the unmodified program P_w. The goal of this attack is to thwart a definitive (courtroom) identification of the embedded watermark, by constructing a false decoder $d' \neq d$, and perhaps a false input $k' \neq k$, such that the watermarked program P_w seems to bear a spurious watermark $w' \neq w$ claimed by the attacker.

A novel feature of our tamper-proofing technique is that it embeds the watermark-decoding function d into the watermarked program. This will enable a defender's expert witness to persuasively argue that d is the appropriate decoder for the watermark in P_w. An attacker cannot plausibly lay claim to an arbitrary decoder d', although they might successfully make a spurious claim to some d' that is a subroutine of our d. Thus the protocol attacker must find a d' very similar to the true decoder d, and an input sequence k', with the property that $d'(\mathcal{X}(P_w, k')) = w' = p'q' \neq w$ for large primes p' and q'. The probability of success in this endeavour is exceedingly remote, unless the attacker has very detailed knowledge of how watermarks are built dynamically by P_w. Even with this knowledge, the attack is very unlikely. Thus, unless a defender is foolish or careless enough to reveal the secret primes p and q used to construct watermark w, they can be confident of surviving a protocol attack on P_w in the courtroom.

5 Conclusion

In this paper we have formally defined the problem of tamperproofing a software watermark. We have described an implementation of our "constant encoding" solution to this problem. We have briefly described the results of our feasibility study (consisting of extensive Monte Carlo experimentation and some lemma-proving), showing that commonly occurring constants in computer programs can indeed be replaced automatically, by a random selection from a very wide range of possible function calls and arguments. Our security analysis indicates that our solution would indeed be tamperproof to a reverse engineer who can perform only a static analysis; and that a long series of dynamic analyses would be required to safely remove the watermark. We have argued, somewhat less convincingly, that our solution would also be resistant to pattern-matching attacks. We expect to be able to mount a more convincing argument after we have completed a prototype full implementation. Then we can tamperproof a few sample watermarked programs, and we can ask some talented reverse engineers to try to remove or modify these watermarks without damage to program correctness. We believe that such human-factor experiments are required to properly evaluate the security of any real system; theoretical arguments in favour of security are always limited by *ad hoc* assumptions about the skill, knowledge and resources of the attacker.

References

Ball, T. & Larus, J. R. (1996), Efficient path profiling, *in* 'Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29)', pp. 46–57.

Collberg, C. & Thomborson, C. (1999), Software watermarking: Models and dynamic embeddings, *in* 'Principles of Programming Languages (POPL'99)', San Antonio, TX, pp. 311–324.

Collberg, C., Thomborson, C. & Low, D. (1998), Manufacturing cheap, resilient, and stealthy opaque constructs, *in* 'Principles of Programming Languages (POPL'98)', San Diego, CA, pp. 184–196.

He, Y. (2002), Tamperproofing a software watermark by encoding constants, Master's thesis, Comp. Sci. Dept., Univ. of Auckland.

Holmes, K. (1994), 'Computer software protection', US Patent 5,287,407. Assignee: International Business Machines.

Kreher, D. & Stinson, D. (1999), *Combinatorial Algorithms*, CRC Press LLC.

Moskowitz, S. A. & Cooperman, M. (1998), 'Method for stega-cipher protection of computer code', US Patent 5,745,569. Assignee: The Dice Company.

Nagra, J., Thomborson, C. & Collberg, C. (2002), A functional taxonomy for software watermarking, *in* M. Oudshoorn, ed., 'Proc. 25th Australasian Computer Science Conference 2002', ACS, pp. 177–186.

Palsberg, J., Krishnaswami, S., Kwon, M., Ma, D., Shao, Q. & Zhang, Y. (2001), Experience with software watermarking, *in* 'Proc. 16th Ann. Comp. Security Applications Conf. (ACSAC'00)', IEEE Computer Society, pp. 308–316.

RSA Laboratories (2002), 'Public-key cryptography standard (PKCS) #1 v2.1', ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf.