Dither Modulation Watermarking of Dynamic Memory Traces

Alan J. Larkin*, Félix Balado, Neil J. Hurley**, and Guenolé C. M. Silvestre**

Department of Computer Science,
University College Dublin,
Belfield, Dublin 4,
Ireland.
{alan, fiz, neil.hurley, guenole.silvestre}@ihl.ucd.ie
http://www.ihl.ucd.ie

Abstract. We describe a dynamic software watermark embedded in the memory trace of an executing Java program. Our approach is a generalisation of the spread–transform watermarking technique developed for use in the multimedia domain. We show how the spread–transform paradigm enables the embedding of dither modulation watermarks in a Java program and report its robustness to realistic additive noise attacks.

1 Introduction

Recent years have witnessed an hitherto unseen level of concern about the issue of software piracy and intellectual property (IP) protection, among commercial and non–commercial software developers alike.

The advent of the Internet has radicalised software business models. It has now become common practice for vendors to make *trial* versions of even the most costly and sophisticated software freely available for download. Although such trial programs are typically time limited or partially disabled, they necessarily contain most of the developer's code and IP. This constitutes a major risk to the developer as this software is now vulnerable to attack by *crackers*, who attempt to disable the protection techniques, and plagiarism by competitors. Furthermore, the Internet has made possible the rapid mass distribution of derived or cracked software, enabling virtually any computer user to become a significant threat to even heavyweight developers.

Another related, although distinct, contributing factor has been the remarkable rise of the Java programming language. One of the principal reasons for Java's popular success has been its *"write once, run anywhere"* paradigm. In attaining this goal the language developers devised a compiled Java format (bytecode), which is effectively isomorphic to the original source code. As a consequence, Java software is easily reverse engineered, which makes it an insecure medium for algorithms and other IP.

* Supported by Irish Research Council for Science Engineering and Technology grant RS/2002/798–2

** Supported by Enterprise Ireland Basic Research Grant SC/2002/178

A final development which is threatening software creators is the marked increase in interest in decompilation and other reverse engineering tools and techniques. As reverse engineering becomes ever more sophisticated, software developers are finding it increasingly difficult to conceal their IP, even by choosing to work in languages that are traditionally more difficult to reverse engineer than Java, such as C++.

Many tools exist for combating these problems, from the legal to the technical. One technical solution is remote execution, whereby the most intellectually valuable sections of a program reside only on secure servers [19]. A registered client program utilises the server as an oracle; sending input and receiving output, but never being exposed to the internals of the computation. Such an approach can be highly effective for detecting illegal software use (by an unlicensed user) and preventing IP theft. However remote execution is generally unattractive to legitimate customers for efficiency, reliability and privacy reasons.

Other approaches have been based on the use of hardware tokens such as *dongles*. These devices were typically plugged into a computer's serial port. The correct execution of a program was reliant on the presence of its dongle. More recently this concept has been advanced through the use of smart cards [10], which enable small portions of a program's computation to be performed by the token. Both techniques complicate piracy since the hardware devices are not readily reproducible. However this approach is not compatible with modern lightweight web–based business practices.

The most widely used software based protection technique is *obfuscation* [4] — the process of transforming one program into an equivalent one, which is more difficult to reverse engineer and understand. Obfuscation has the advantage that it is cheap and non–intrusive and can be quite effective. However it is only an impediment to theft. Obfuscation is not useful for identifying a theft, nor resolving the true *owner* of a given piece of software. These are the functions of a *software watermark*.

1.1 Software Watermarking

A software watermark is a piece of information that is embedded within a program. The presence of the watermark does not change the functionality of the program, and, generally, its presence should only be detectable through the use of an authorised detector. When used to encode ownership information, watermarking may be employed to both deter and detect theft of the host program, and in the event of such a theft, help resolve rightful ownership over it. Consequently, a would–be thief is required to destroy any watermarks present in the code they intend to steal, or risk detection and prosecution. Software watermarks must be resilient to such deliberate *attacks*.

Digital watermarking originally evolved in the multimedia domain. It has been the subject of much research and has become a formal and mature discipline [12, 14]. Multimedia, such as images and audio, have natural representations

as vectors in numeric spaces. These vectors can be manipulated to encode a watermark without introducing any humanly perceptible changes.

The application of watermarking protection techniques to software is a relatively recent development, and although a number of techniques have been devised, the art remains in its infancy. Despite being conceptually similar, multimedia and software watermarking are in practice quite different. The smooth and natural mappings from objects to vectors employed in the multimedia domain, are not naturally present in the software domain. Instead, software watermarks tend to be embedded at a more abstract level; for example in the topology of dynamically created heap structures [3], or in a program's control–flow–graph [18].

A limited amount of work has been carried out in which a concerted effort is made to view the software watermarking problem as one to which multimedia solutions can be adapted [16, 7, 6, 15]. In each of these cases, an additive *spread–spectrum* algorithm [5] was applied to a feature vector extracted from the host program.

In this paper we too seek to exploit results from the multimedia domain to improve software watermarking, however our approach differs significantly from the others. We describe a novel approach for watermarking the collective run–time memory consumption of the set of Java methods comprising a program. In our case, the feature vector extraction process can be seen as an extension of the *spread–transform* technique (a more general method of spreading watermark information over a host signal than spread–spectrum) that is frequently employed on multimedia but, to date, has never applied to software. To this feature vector we apply a quantisation based watermarking algorithm [1]. Again the advantages of such algorithms over simple additive ones have not yet been exploited by the software watermarking community. We describe in particular how *dither modulation* watermarking is well suited to our problem.

This paper is organised as follows; In Section 2 we introduce the notion of a memory trace of a Java program, and show how the general process of watermarking a program's memory consumption may be viewed in terms of the spread–transform. Section 3 describes dither modulation watermarking, in particular its application to memory traces, and in Section 4 we present a statistical attack model against which we evaluate our proposal.

Notation: In this paper we will employ the following notational conventions;

- Lowercase bold roman letters ($\mathbf{x}$) denote vectors. All vectors are assumed to be column vectors. Row vectors are denoted with the use of a ^{T} superscript, meaning *transpose.* The ith element of a vector $\mathbf{x}$ is denoted by x_i.
- Matrices are set in bold uppercase roman letters ($\mathbf{A}$). The (i, j) element of a matrix $\mathbf{A}$ is denoted by a_{ij}.
- $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ represent the set of natural, integer and real numbers respectively. An additional scalar superscript, for example $\mathbb{R}^N$, denotes the N–

dimensional cartesian product of that set. Unless otherwise specified, a $^+$ superscript denotes the subset of positive numbers.
- $\lceil\cdot\rceil$ denotes rounding up to the next integer. $\|\cdot\|$ denotes a Euclidean norm. $\mathrm{E}\{\cdot\}$ denotes expectation.

2 Memory Trace Spread–Transform Watermarking

We define the N–dimensional memory trace (MT) of a Java program as the N–element vector corresponding to the total dynamic memory allocation performed during each of the N equal length time intervals $T_1, \ldots, T_N$, of an execution of that program;

$$x_i^{\mathrm{ST}} = \text{memory allocated during } T_i.$$

This signal is of course inherently *keyed* upon the program input, or more generally upon the context of the execution.

If the program is comprised of K methods $\{m_1, \ldots, m_K\}$, we can consider the dynamic trace of the sequence of method invocations, and express x_i^{ST} as

$$x_i^{\mathrm{ST}} = \sum_{j=1}^{K} a_{ij} \overline{x}_{ji},$$

where a_{ij} denotes the number of invocations of method m_j during interval T_i, and $\overline{x}_{ji}$ denotes the *average* number of bytes consumed over all executions of m_j during interval T_i. Define $\mathbf{A} = \{a_{ij}\} \in \mathbb{N}^{N,K}$ to be the *method execution frequency matrix* and $\overline{\mathbf{X}} = \{\overline{x}_{ji}\} \in \mathbb{R}^{K,N}$. The preceeding equation then becomes

$$\mathbf{x}^{\mathrm{ST}} = \mathrm{diag}(\mathbf{A}\overline{\mathbf{X}}).$$

Modulating a watermark vector $\mathbf{w}$ onto the MT vector $\mathbf{x}^{\mathrm{ST}}$ leads to the problem of solving

$$\mathbf{x}_w^{\mathrm{ST}} = \mathbf{x}^{\mathrm{ST}} + \mathbf{w} \quad (1)$$

$$= \mathrm{diag}(\mathbf{A}(\overline{\mathbf{X}} + \Delta\overline{\mathbf{X}}), \quad (2)$$

for matrix $\Delta\overline{\mathbf{X}}$, whose (i, j) component represents the amount by which the *average* memory consumption of m_j during T_i must be modified in order to embed $\mathbf{w}$.

Note that modifying a method's memory consumption by an amount that varies with the current interval of execution requires complicated dynamic control flow analysis and/or unstealthy modifications (conditional statements pertaining to the context in which the method is being invoked) to be applied to the program source. To avoid these problems we seek instead a solution which is *context–free*, that is, one in which the overhead added to each method's memory consumption is independent of the time interval. Thus we reformulate (2) as

$$\mathbf{x}_w^{\mathrm{ST}} = \mathrm{diag}(\mathbf{A}\overline{\mathbf{X}}) + \mathbf{A}\Delta\overline{\mathbf{x}} \quad (3)$$

where $\Delta\overline{x}_j$ represents the *constant* memory overhead that must be added to method m_j.

From (3) it becomes clear at this point that watermarking the MT vector $\mathbf{x}^{\text{ST}}$ with watermark vector $\mathbf{w}$, is analogous to Chen and Wornell's [1] spread–transform (ST) watermarking of $\overline{\mathbf{X}}$, in which the ith element of the vector $\mathbf{A}\Delta\overline{\mathbf{x}}$ is embedded in the linear projection of $\overline{\mathbf{X}}$ onto the ith row of $\mathbf{A}$.

An important difference between standard ST and our application is the structure of $\mathbf{A}$. In standard ST, $\mathbf{A}$ has the form,

$$\mathbf{A} = \text{block diag}(\mathbf{t}^T) = \begin{pmatrix} \mathbf{t}^T & & 0 \\ & \ddots & \\ 0 & & \mathbf{t}^T \end{pmatrix}, \tag{4}$$

for some *spreading–vector* $\mathbf{t} \in \mathbb{R}^{\lceil \frac{K}{N} \rceil}$. In our case, the ST must be generalised to projection matrices without this particular shape. The significance of this difference becomes apparent when solving (3) for $\Delta\overline{\mathbf{x}}$.

When $\mathbf{A}$ is of the form in Equation (4), the inversion of the projection onto $\mathbf{A}$ can be performed for each row of (3) independently of the others, making the computation trivial. In our generalised case however, the projection must be reversed for all of the rows *simultaneously.*

$$\begin{aligned} \Delta\overline{\mathbf{x}} &= \mathbf{A}^{+}(\mathbf{x}_w^{\text{ST}} - \text{diag}(\mathbf{A}\overline{\mathbf{X}})) \\ &= \mathbf{A}^{+}(\mathbf{x}^{\text{ST}} + \mathbf{w} - \text{diag}(\mathbf{A}\overline{\mathbf{X}})) \\ &= \mathbf{A}^{+}(\text{diag}(\mathbf{A}\overline{\mathbf{X}}) + \mathbf{w} - \text{diag}(\mathbf{A}\overline{\mathbf{X}})) \\ &= \mathbf{A}^{+}\mathbf{w}, \end{aligned} \tag{5}$$

where $\mathbf{A}^{+}$ denotes the Moore–Penrose pseudo–inverse of $\mathbf{A}$, given by $\mathbf{A}^{+} \triangleq \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}$.

2.1 Watermarking Embedding

The above discussion suggests the following general procedure for embedding a ST watermark in the MT of a Java program;

1. Use some *profiling* process to extract the memory trace $\mathbf{x}^{\text{ST}}$ and method execution frequency matrix $\mathbf{A}$ from the program as it executes under some *special* input (the key)[1].
2. Obtain a watermark $\mathbf{w}$ for the given information to be embedded, and compute $\Delta\overline{\mathbf{x}}$ (5).
3. Modify each of the program's K methods, so that the new method m_j allocates, on average, $\Delta\overline{x}_j$ more bytes on the heap per execution than the original method.

[1] Note that (5) is independent of $\mathbf{x}^{\text{ST}}$, and as such this technique does not strictly require access to the original memory trace, however in the following section we exploit access to it for watermark message encoding.

Several of the practical issues which arise with this embedding process are worth noting at this point;

1. Real–valued solutions to (5) are not desirable, since memory cannot be allocated in fractions of bytes in Java code.
2. Furthermore, the solution space is made discrete by the target Java Virtual Machine's (JVM) memory manager. For example, on the Sun JVM [11], memory is allocated in blocks of 8 bytes with a minimum of 2 blocks, implying $\Delta\overline{\mathbf{x}} = 8\mathbf{y}, \mathbf{y} \neq \mathbf{1} \in \mathbb{Z}^m$.
3. Negative elements in $\Delta\overline{\mathbf{x}}$ imply that the corresponding method must be optimised to use a certain number of bytes less memory than before. Performing such targeted optimisation is a hard problem.
4. Finally, a method's memory consumption may be affected by its context of execution, requiring care to be taken to ensure method modifications have the desired effects.

The first two of these issues can be addressed by reformulating (5) as an optimisation problem,

$$\min_{\Delta\overline{\mathbf{x}}} \|\mathbf{A}\Delta\overline{\mathbf{x}} - \mathbf{w}\| \quad \text{s.t.} \quad \Delta\overline{\mathbf{x}} = 8\mathbf{y}, \quad \mathbf{y} \neq \mathbf{1} \in \mathbb{Z}^m, \tag{6}$$

thus suitably constraining the solution space. Point 4 is most simply dealt with by making all modifications in the first basic block[2] of each method's bytecode, thus removing the effects of execution context, although more sophisticated, stealthy approaches may be developed. We postpone discussion of the remaining issue of negative solutions until the next section.

2.2 Watermarking Detection

Having developed an embedding process we now turn our attention to decoding. In ST watermarking, watermark decoding must be performed on the projection of the received signal $\widehat{\overline{\mathbf{X}}}$, a distorted version (due to attacks or other interference) of $\overline{\mathbf{X}}$;

$$\widehat{\mathbf{x}}^{\text{ST}} = \operatorname{diag}(\mathbf{A}\widehat{\overline{\mathbf{X}}}). \tag{7}$$

Recall that, in practice, this amounts to extracting the MT $\widehat{\mathbf{x}}^{\text{ST}}$ from the received program via profiling of an execution under the key program input. Once obtained, the MT is subjected to the detection routine counterpart of the watermark message coding method used at the embedder.

Equation (7) fails to highlight another important difference between our application of ST to software and its standard use in the multimedia domain; the projection matrix $\mathbf{A}$ is part of the object being transmitted, and as such may be subjected to distortion. The previous equation should read,

$$\widehat{\mathbf{x}}^{\text{ST}} = \operatorname{diag}(\widehat{\mathbf{A}}\widehat{\overline{\mathbf{X}}}),$$

[2] A basic block is a sequence of one or more consecutive instructions having only one entry point and one exit point.

where $\widehat{\mathbf{A}}$ denotes the received program's method execution frequency matrix. Even an undistorted program is likely to yield mildly differing method execution frequency matrices from one execution to the next due to chaotic processes such as IO blocking, or thread interleaving within the JVM. However we assume that the time interval is suitably large (N is small) so that $\mathbf{A} \simeq \widehat{\mathbf{A}}$.

3 Dither–Modulation in the Memory Trace Domain

In Equation (1), a vector $\mathbf{w}$ was modulated onto the ST of the host signal. At this point any watermarking algorithm may be applied to determine the value of $\mathbf{w}$. Similarly to previous work in the multimedia domain [1], we employ the Dither Modulation (DM) algorithm. In this section we describe the approach and discuss its particular merits with respect to Java MT watermarking.

DM is a form of Quantisation Index Modulating (QIM) watermark [1], a class of algorithms in which information is embedded via the choice of one *quantiser* from a set, and the application of that quantiser to the original signal.

A scalar quantiser $\mathcal{Q}$ is a mapping from a one dimensional space, to a discrete subset of that space. For example,

$$\mathcal{Q} : \mathbb{R} \to \{c_i \, : \, c_i \in \mathbb{Z}\}.$$

Normally, a minimum–distance mapping is employed.

In QIM, a set of quantisers $Q = \{\mathcal{Q}_1, \ldots, \mathcal{Q}_l\}$ is defined. At the embedder, the watermark message M is encoded as a sequence $M \equiv (M_1, M_1, \ldots, M_N) \in \{0, \ldots, l-1\}^N$. The ith element of the original signal, in our case $\mathbf{x}^{\text{ST}}$, is then quantised using the quantiser in Q indexed by M_i;

$$x_{w_i}^{\text{ST}} = \mathcal{Q}_{M_i}\left(x_i^{\text{ST}}\right).$$

In conjunction with Equation (1) this implies that

$$w_i = \mathcal{Q}_{M_i}\left(x_i^{\text{ST}}\right) - x_i^{\text{ST}}. \tag{8}$$

This encoding of the watermark message is then added to the projected signal in (1).

During the (blind) watermark extraction phase, the elements of the signal received at the decoder $\widehat{\mathbf{x}}_w^{\text{ST}}$, are quantised using each quantiser in Q. The received message $\widehat{M}$ is reconstructed from the indices of the sequence of quantisers which contain the reconstruction points closest (in a Euclidean sense) to the elements of $\widehat{\mathbf{x}}_w^{\text{ST}}$;

$$\widehat{M} = \left(\arg\min_k \left\| \mathcal{Q}_k\left(\widehat{x}_{w_i}^{\text{ST}}\right) - \widehat{x}_{w_i}^{\text{ST}} \right\|\right)_{i=1}^{N}.$$

The Binary DM (Fig. 1) algorithm which we utilise in the remainder of this paper is a specialisation of QIM, in which two uniform quantisers $Q = \{\mathcal{Q}_0, \mathcal{Q}_1\}$ are employed. $\mathcal{Q}_0$ and $\mathcal{Q}_1$ are both of step size $\Delta \in \mathbb{R}$ and are mutually shifted, or *dithered*, by $\Delta/2$.

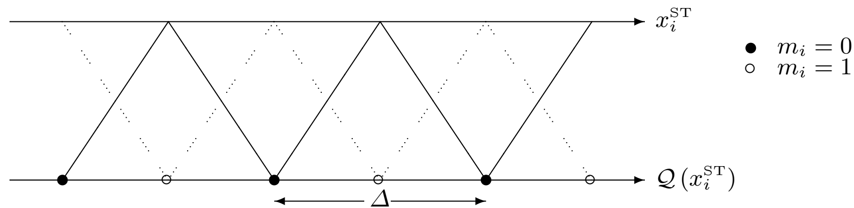

Fig. 1: Binary Dither Modulation in the ith dimension. The ith element of the original signal $\mathbf{x}^{\text{ST}}$ (upper line) is *quantised* to either a • or a ◦ (lower line) if $M_i = 0$ or $M_i = 1$, respectively. With a quantiser step size of Δ, the maximum distortion that it will be necessary to introduce per dimension of $\mathbf{x}^{\text{ST}}$ is $\frac{\Delta}{2}$. After embedding, the maximum tolerable amplitude distortion per dimension before an error is introduced is $\frac{\Delta}{4}$.

Our motivation for selecting the DM algorithm over spread–spectrum (as has been the trend in the software domain) is two–fold. The first is an inherent property of QIM watermarking termed *host signal interference rejection.* QIM's exploitation of knowledge of the host signal $\mathbf{x}^{\text{ST}}$ at the encoder (c.f. footnote [3]), prevents the host signal from acting as interference in the transmission of the watermark message from the embedder to the detector, thus improving the watermark's robustness to distortion/capacity. The second is a practical advantage which we contrive from the properties of the MT domain.

First we note that quantising to *any* reconstruction point in the appropriate quantiser is enough to encode a given watermark element. For image and audio data, relatively low limits on the acceptable level of distortion introduced due to watermarking and attacking alike, are naturally defined by the human sensory systems. Consequently, quantising to the *nearest* quantisation point is important in order to preserve the imperceptibility of the watermark. In the MT domain however, the imperceptibility of the embedded information is not of overriding importance. Although it may be considered undesirable for a watermark to significantly increase the memory footprint of a given piece of software, the principal concern and only strict requirement is the preservation of the functionality of the program.

This realisation enables us to address the outstanding practical issue raised in the previous section; that of negative values in $\Delta\overline{\mathbf{x}}$. By always quantising *up* to the nearest quantisation point, we can ensure that non–negative solutions to Equation (6) can be generated. Thus our DM quantisers are of the form;

$$\mathcal{Q}_0(x) = \left\lceil \frac{x}{\Delta} \right\rceil \Delta \quad \text{and} \quad \mathcal{Q}_1(x) = \left(\left\lceil \frac{x - \frac{\Delta}{2}}{\Delta} \right\rceil + \frac{1}{2} \right) \Delta. \tag{9}$$

Additionally we note that when Δ is chosen to be divisible by 16, simple manipulation of (6) using (8) and (9) reduces the problem to an optimisation of a natural number linear system, thus permitting natural solutions.

4 Experimental Results

4.1 Experimental Framework

Implementation of a system illustrated in Fig. 2 has been undertaken in order to facilitate experimentation and analysis of the proposed watermark. The embedding process consists of three principle stages; 1) feature extraction or profiling, 2) formulation and solution of Equation (6), and 3) realisation of the required program modifications.

1. The extraction of the MT vector and method execution frequency matrix is achieved through the use of a purpose built profiling tool. Constructed as client–server pair, our Java profiler is built upon the Java Virtual Machine Profiling Interface (JVMPI) [9]. The profiling client resides in the same process as the JVM. User–specified statistics of interest are gathered on a per–thread basis, and with minimal client–side processing are sent to the server. This minimal processing, per–thread profiling ensures the least possible distortion of the extracted signals due to the presence of the profiler. The server receives the raw data, re–synchronises it, and processes it in a user–specified manner into a database describing the execution of the program. Generating such a database enables the user to create many different views of a single execution.
2. For the MT watermarking application, the features of interest are the MT itself and the method execution frequency matrix. These are extracted during stage 2, and are used to formulate a representation of Equation (6). This optimisation problem is non–trivial, and so we employ the NEOS Server [8] to generate solutions. NEOS is a free web–server which offers access to a large number of mathematical optimisation suites. Specifically, we send our problem to NEOS for solution using the XPRESS–MP Mixed Integer Linear Programmer [13]. (Note that we are restricted to solving Equation (6) as an L_1–norm minimisation problem, rather than the more desirable L_2–norm). Frequently, exact solutions to our problems do not exist, and so there is normally some embedding error inherent in our process. Indeed, on occasion, no solutions are found or the problem is determined to be infeasible. In such cases we have no recourse other than to re–profile under a different program input.
3. Assuming an acceptable solution can be computed, we proceed with the final step, modifying the program's methods so that they allocate extra memory as dictated by NEOS. This can be as trivial as the insertion of superfluous `byte` array allocation instructions at the start of each method, although more sophisticated approaches involving the use of opaque predicates could and should be used. In what follows this last step is simulated.

We now report results from simulated embedding and attacking of the watermarking scheme. Our early results are restricted to simulations on a single Java program shown in Table 1. Experiments were performed on the Sun JVM

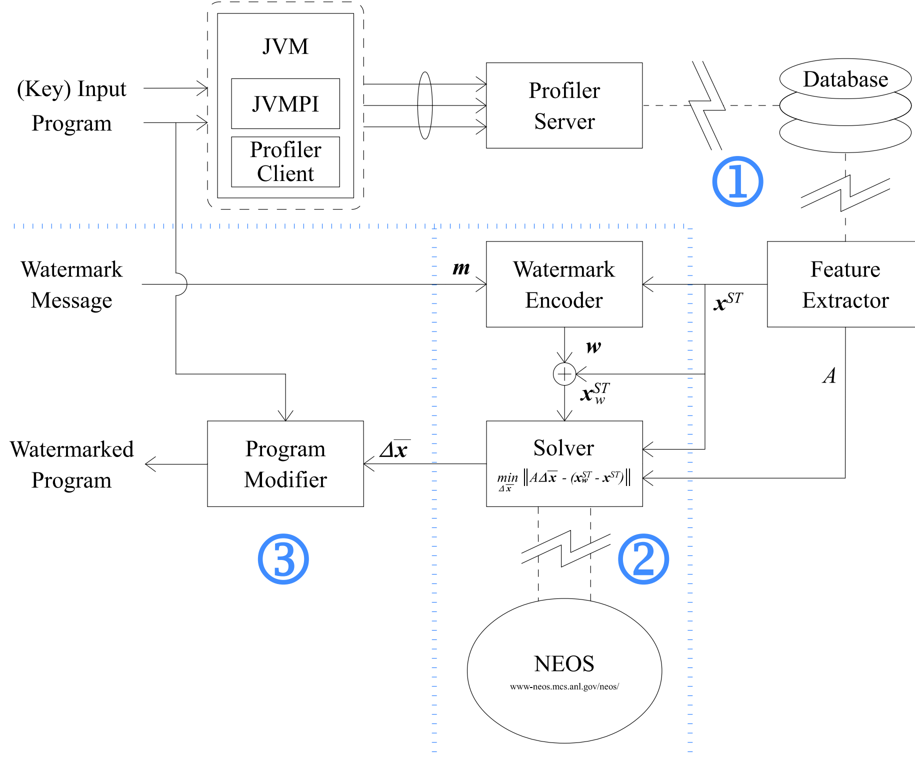

Fig. 2: Memory Trace Watermarking Framework.

Table 1: Test Program.

Name:	Java Tree Builder (JTB) [17]
Description:	JavaCC Grammar annotater, composed of 1447 methods
Input:	Java1.2-a.jj

(version 1.4.2–02), running under GNU Linux on a dual Intel Xeon 2.8 GHz machine, with 2.5 Gb RAM.

4.2 Attack Model

Proper analysis of a watermarking scheme's robustness requires a model of the attacks to which the watermarked signal may be subjected. In this paper we employ an additive noise model derived from experimentation on a set of automated attacks from the SandMark system [2] (Table 2).

The amplitude distortions introduced into the watermark channel (the average method memory consumptions, $\overline{\mathbf{X}}$) by application of these attacks to the test

Table 2: Considered Attacks from the SandMark System.

Name	Description
Array Folder	"Folds" a 1–dimensional array into a multi–dimensional array.
Array Splitter	Splits 1–dimensional array fields into 2 array fields.
Block Marker	Randomly marks all basic blocks in a program with either a 0 or 1.
Constant Pool Reorderer	Reorders the constants in the constant pool and assigns random indices to them.
False Refactor	Two classes `C1` and `C2` that have no common behavior are refactored forming a class `C3`.
Field Assignment	Inserts bogus fields into a class and makes assignments to that field in specific locations throughout the code.
Insert Opaque Predicates	Inserts an opaque predicate into every boolean expression.
Integer Array Splitter	Splits a local variable array into two arrays.
Overload Names	Renames methods so that as many as possible have the same name.
Publicize Fields	Makes the fields of a class public.
Rename Registers	Renames local variables to random identifiers.
Reorder Parameters	Shuffles the argument orders for all methods.

program were measured. Under the assumption that these noise samples are independent and identically distributed, the Central Limit Theorem predicts that they should follow Normal distributions. Empirical evidence in fact shows that they are drawn from peaked distributions. We conclude that the independence assumption does not hold, due to correlations existing amongst both the rows of the execution frequency matrix (as a result of *the principle of locality*), and its columns (as a result of inter–method dependencies).

Using maximum likelihood estimation, the noise signals due to the considered attacks were in fact all found to be well approximated by the Laplacian distribution;

$$f_X(x; \mu, \lambda) = \frac{\lambda}{2} e^{-\lambda|x-\mu|}.$$

The estimated parameters for each attack are listed in Table 3.

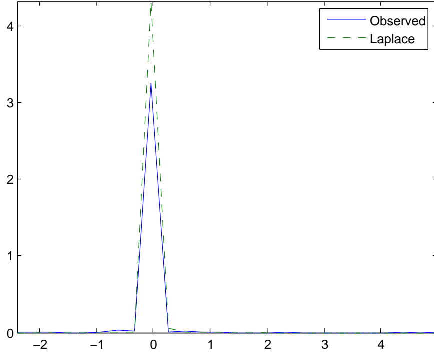

Fig. 3: Laplacian Model of Array Splitter Attack.

Attack	μ	λ
Array Splitter	81.1735	0.0020650
Constant Pool Reorderer	51.3785	0.0026051
Overload Names	-23.1289	0.0030382
Array Folder	48.5469	0.0030583
Integer Array Splitter	58.8219	0.0036680
False Refactor	57.0828	0.0037142
Block Marker	56.2616	0.0037221
Insert Opaque Predicates	74.3434	0.0041431
Publicize Fields	60.3178	0.0045263
Field Assignment	64.1785	0.0046936
Reorder Parameters	57.2756	0.0047845
Rename Registers	57.9861	0.0048702

Table 2: Parameters of Laplacian Model of SandMark Attacks.

4.3 Robustness

7–4 Hamming error–correction–coded (ECC) watermarks were embedded in the $N = 32$–dimensional MT of the test program[3]. At this dimension, the correlation between the original matrix $\mathbf{A}$ and each of the attacked frequency matrices $\widehat{\mathbf{A}}$, bar one[4], was found to be in the range [0.899, 0.993]. In these experiments $N = 32$ was thus considered sufficiently small so that $\mathbf{A} \simeq \widehat{\mathbf{A}}$ (Section 2.2).

We measure the robustness of the transmission of the watermark message from the embedder to the detector as the proportion of erroneously transmitted bits, or the *bit–error–rate* (BER). Fig. 4 shows the simulated BER against the *watermark–to–noise–ratio* (WNR) for JTB under attack with Laplacian noise. The WNR in decibels is defined as,

$$\mathrm{WNR} = 10 \log_{10} \frac{\mathrm{E}\{\|\mathbf{A}\Delta\overline{\mathbf{x}}\|^2\}}{\mathrm{E}\{\|\mathbf{A}\mathbf{d}\|^2\}},$$

where $\mathbf{d}$ is the noise vector, and provides a fair measure of the strength of a watermark in relation to an attack. Note that we compute the WNR in the projection domain (the runtime memory consumption per interval), arguing that for our application it is in this domain that the (im)perceptibility of the watermark should be measured.

...

[3] For these experiments, $N = 32$ implied interval widths of 485 ms, real time.
[4] Correlation after the Array Splitter attack was found to be 0.795.

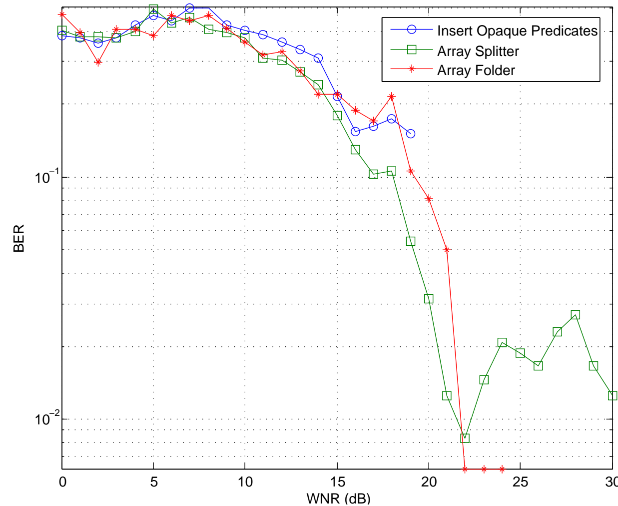

Fig. 4: Bit–Error–Rate *vs.* WNR for JTB Attacked with Laplacian Noise.

5 Summary and Conclusions

In this paper we have described a new form of dynamic Java watermarking, which is unique both in terms of the feature vector used to carry the watermark, and in its application of ST–DM principles. . . .

Additionally, we have modelled of a set of attacks to which this watermark may be subjected. Within the software watermarking community the term *attack model* is frequently used in an informal manner, often being applied to what are little more than lists of program transformations. Such *models* are of poor use for forecasting the performance of a watermarking system. Our model on the other hand, is mathematical, and has enabled us to make predictions about the robustness of our watermark, and argue that for a watermark of reasonable energy, perfect transmission is possible even in the face of these attacks.

This model does however remain modest. As yet we have considered only a narrow band of the full spectrum of automated transformations to which software may be subjected. Further experimentation may expose the additive noise model to be unsuitable for describing processes such as optimisation, decompilation, or multiple obfuscations applied in series. These attacks may prove to be more *desynchronsing* than additive in nature.

Future work must focus on expansion of our attack model, and more complete evaluation of this watermark's robustness. Beyond this, an interesting direction for future research is the embedding of the watermark in a frequency domain, such as the Fourier Transform, of the MT. Again such techniques have long been employed on multimedia, but are yet to make the crossover to software. We

hypothesise that embedding in the low frequency components (corresponding to the global shape of the MT) of such a transformed signal, will improve resilience to attacks, which appear to introduce noise primarily at the local level.

Acknowledgements

The authors wish to gratefully acknowledge the support of Enterprise Ireland (grant SC/2002/178) and the Irish Research Council for Science Engineering and Technology (grant RS/2002/798–2).

We also wish to thank Julian Keenaghan for his helpful contribution.

References

1. B. Chen and G. W. Wornell. Quantization Index Modulation: A Class of Provably Good Methods for Digital Watermarking and Information Embedding. In *ISIT: Proceedings IEEE International Symposium on Information Theory, sponsored by The Information Theory Society of The Institute of Electrical and Electronic Engineers*, volume 47, pages 1424–1443, May 2000.
2. C. S. Collberg. SandMark *User's Guide*, Jan 2003.
3. C. S. Collberg and C. D. Thomborson. Software Watermarking: Models and Dynamic Embeddings. In *Principles of Programming Languages 1999, (POPL'99)*, pages 311–324, Jan 1999.
4. C. S. Collberg, C. D. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report TR98–148, Department of Computer Science, The University of Auckland, Jul 1997.
5. I. J. Cox, J. Kilian, T. Leighton, and T. Shamoon. A Secure, Robust Watermark for Multimedia. In *Workshop on Information Hiding*, pages 175–190, May 1996.
6. D. M. Curran, N. J. Hurley, and M. Ó Cinnéide. Securing Java Through Software Watermarking. In *Principles and Practice of Programming in Java (PPPJ'03)*, Jun 2003.
7. G. Hachez. *A Comparative Study of Software Protection Tools Suited for E–Commerce with Contributions to Software Watermarking and Smart Cards*. PhD thesis, Universite Catholique de Louvain, Mar 2003.
8. Argonne National Laboratory. NEOS Optimization Server, __.
9. S. Liang and D. Viswanathan. Comprehensive Profiling Support in the Java Virtual Machine. In *5th USENIX Conference on Object–Oriented Technologies and Systems (COOTS'99)*, May 1999.
10. A. Maña and E. Pimentel. An Efficient Software Protection Scheme. In M. Dupuy and P. Paradinas, editors, *Trusted Information: The New Decade Challenge, IFIP TC11 Sixteenth Annual Working Conference on Information Security (IFIP'01)*, volume 193 of *IFIP Conference Proceedings*, pages 385–402. Kluwer, Jun 2001.
11. Sun Microsystems. Java Virtual Machine, __.
12. M. Miller, I. J. Cox, J. P. Linnartz, and T. Kalker. A Review of Watermarking Principles and Practices. *Digital Signal Processing in Multimedia Systems*, 1999.
13. Dash Optimization. Xpress–MP, __.
14. F. A. P. Peticolas, R. J. Anderson, and M. G. Kuhn. Information Hiding — A Survey. In *Proceedings of the IEEE*, volume 87, pages 1062–1078, Jul 1999.

15. T. Sahoo and C. Collberg. Software Watermarking in the Frequency Domain: Implementation, Analysis, and Attacks. Technical Report TR04–07, Department of Computer Science, The University of Arizona, Mar 2004.
16. J. P. Stern, G. Hachez, F. Koeune, and J.–J. Quisquater. Robust Object Watermarking: Application to Code. In A. Pfitzmann, editor, *Information Hiding: 3rd International Workshop (IH'99)*, volume 1768 of *Lecture Notes in Computer Science (LNCS)*, pages 368–378, Germany, 2000. Springer.
17. UCLA Compilers Group. Java Tree Builder, __.
18. R. Venkatesan, V. Vazirani, and S. Sinha. A Graph Theoretic Approach to Software Watermarking. In I. S. Moskowitz, editor, *Information Hiding Workshop (IH'01)*, volume 2137 of *Lectures Notes in Computer Science (LNCS)*, pages 157–168, Pittsburgh, PA, USA, 2001. Springer–Verlag.
19. X. Zhang and R. Gupta. Hiding Program Slices for Software Security. In *International Symposium on Code Generation and Optimization (CGO'03)*, page 325, Mar 2003.