Self-validating Branch-Based Software Watermarking

Ginger Myles and Hongxia Jin

Computer Science Division, IBM Almaden Research Center,
San Jose, CA 95120
{gmyles, jin}@us.ibm.com

Abstract. Software protection is an area of active research in which a variety of techniques have been developed to address the issue. Examples of such techniques are software watermarking, code obfuscation, and tamper detection. In this paper we present a novel dynamic software watermarking algorithm which incorporates ideas from code obfuscation and tamper detection. Our technique simultaneously provides proof of ownership and the capability to trace the source of the illegal redistribution. It additionally provides a solution for distributing pre-packaged, fingerprinted software which is linked to the consumer. Our technique is specific to programs compiled for the x86 Intel architecture, however, we have proposed an extension for use on Java bytecode.

1 Introduction

The problem of protecting software from illegal copying and redistribution has been the focus of considerable research motivated by billions of dollars in lost revenue each year [1]. The growing concern regarding software piracy can be attributed to a variety of factors such as the distribution of software in architectural neutral formats and the ease of sharing over the Internet. In previous years piracy was limited by the necessity to physically transfer a piece of software on a floppy disc or CD-ROM. With the increases in bandwidth, physical transfer is no longer necessary.

In the unfortunate event that software is illegally redistributed or an important algorithmic secret is stolen, an owner would like to be able to take action against the theft. This requires demonstration of ownership and/or identification of the source of the illegal redistribution. A technique which enables such action is *software watermarking*.

Software watermarking is used to embed a unique identifier in a piece of software in order to encode identifying information. While this technique does not prevent piracy, it does provide a way to prove ownership of pirated software. In some cases it is even possible to identify the original purchaser. However, for software watermarking to be useful it must be resilient against a variety of attacks, e.g. semantics-preserving code transformations and program analysis tools.

In this paper we propose a novel dynamic software watermarking algorithm, *branch-based watermarking*, which incorporates ideas from code obfuscation (to aid in preventing reverse engineering) and software tamper detection (to thwart

M. Barni et al. (Eds.): IH 2005, LNCS 3727, pp. 342–356, 2005.

attacks such as the application of semantics-preserving transformations). The heart of the algorithm is centered around redirecting branch instructions to a specifically constructed fingerprint branch function. This function is responsible for computing the program's fingerprint and regulating execution. Through the use of this function automated attacks will result in non-functional software.

The branch-based software watermarking algorithm makes several improvements over previously proposed techniques:

1. Simultaneously provides proof of authorship and the ability to trace the source of the illegal distribution.
2. Demonstrates a significantly higher level of resilience to attack without significant overhead.
3. Provides a means for distributing pre-packaged, fingerprinted software which is linked to the consumer.

2 Software Watermarking

Software watermarking takes the approach of discouraging piracy through a program transformation which embeds a message (the "watermark") into the program. The most basic software watermarking system consists of two functions: $\texttt{embed}(P, w, k) \rightarrow P'$ and $\texttt{recognize}(P', k) \rightarrow w$. Using the secret key k, the `embed`-function incorporates the watermark w into a program P, yielding a new program P'. The `recognize`-function uses the same key k to extract the watermark from a suspected pirated copy.

Each software watermarking algorithm is categorized based on a set of characteristics. These include whether the code is analyzed as a static or dynamic object, the type of recognizer used, the embedding technique, and the type of mark embedded.

Static/Dynamic. Strictly static watermarking algorithms only use features available at compile-time for embedding and recognition. On the other hand, strictly dynamic watermarking algorithms use information gathered during the execution of the program. Abstract watermarking algorithms are neither strictly static or dynamic. Instead, such techniques are static in that recognition does not require execution of the program. However, they are dynamic since the watermark is hidden in the semantics of the program.

Recognition Type. A watermark recognizer is categorized based on the information needed to identify the watermark. Both blind and informed watermarking algorithms require the watermarked program and the secret key to extract the watermark. An informed technique additionally requires an unwatermarked version of the program and/or the embedded mark.

Embedding Technique. To incorporate a watermark, a program has to be manipulated through semantics-preserving transformations. Such transformations can be categorized as follows:

- Reorder or rename code sections.
- Insert new (non-functional and/or never executed) code .

- Manipulate the program's statistical properties such as instruction frequencies.

Mark Type. An *authorship mark* (AM) is a watermark in which the same mark is embedded in every copy of the program. An AM is used to identify the author and is in essence a copyright notice. On the other hand, a *fingerprint mark* (FM) is unique for each copy distributed and is normally used to identify the purchaser. Through the use of a FM it is possible to identify the source of an illegal distribution.

Previous watermarking algorithms use one of the above embedding techniques. In this paper we introduce a new embedding technique in which a section of code is added to the program. The new code both calculates the fingerprint as the program executes and directs program execution.

3 Related Work

A variety of software watermarking algorithms have been proposed. Due to the relative ease of static watermarking there are far more static than dynamic algorithms. A few examples of static watermarking algorithms are those proposed by Venkatesan et al. [2], Stern et al. [3], and Qu and Potkonjak [4]. Venkatesan et al. embed the watermark through an extension to a method's control flow graph. The watermark is encoded in a subgraph which is incorporated into the original graph. Stern et al. modify the instruction frequencies of the original program to embed the watermark. Qu and Potkonjak proposed a very stealthy, but fragile, algorithm which makes use of the graph coloring problem to embed the watermark in the register allocation of the method. In each of these cases, as well as all other static watermarking algorithms, the watermark can be destroyed by basic code optimization or obfuscation techniques.

The first dynamic software watermarking algorithm was proposed by Collberg and Thomborson [5]. This technique embeds the watermark in the structure of a graph, built on the heap at runtime, as the program executes on a particular input. A second dynamic technique proposed by Collberg et al. [6] is path-based and relies on the dynamic branching behavior of the program. To embed the watermark the sequence of branches taken and not taken on a particular input is modified. Two variations for this algorithm were developed to target the varied capabilities of Java bytecode and native executables. A final dynamic technique was developed by Nagra and Thomborson [7]. This technique leverages the ability to execute blocks of code on different threads. The watermark is encoded in the choice of blocks executed on the same thread. Cousot and Cousot [8] developed an abstract watermarking algorithm. The technique uses an abstract interpretation framework to embed a watermark in the values assigned to integer local variables during program execution.

4 Branch Based Software Watermarking

The heart of the branch-based software watermarking algorithm is centered around the use of a branch function specifically designed to generate the pro-

gram's fingerprint as the program executes. If the branch function is properly designed the branch-based algorithm can simultaneously embed authorship and fingerprint marks. Additionally, tamper detection can be incorporated. In the following algorithm description we will provide an example as to how these three features can be incorporated in a single branch function.

The `embed` function for the branch-based algorithm deviates from the standard definition in that it has four inputs and two outputs.

$$\texttt{embed}(P, AM, key_{AM}, key_{FM}) \rightarrow P', FM$$

Using the two secret keys, key_{AM} and key_{FM}, the `embed` function incorporates the authorship mark AM and the fingerprint generating code into the program P, yielding a new program P' and the fingerprint mark FM. Since the algorithm can simultaneously embed an authorship and a fingerprint mark, two secret keys are required. This is in contrast to the usual single key. key_{AM} is tied to the authorship mark and is the same for every copy of the program. key_{FM} is required for the fingerprint mark and should be unique for each copy. A fingerprint mark for a particular instance of a program is based on the fingerprint key and the program execution. Thus, the actual fingerprint mark is generated during embedding and is an output of the `embed` function.

Similarly, the `recognize` function is non-standard with three inputs and two outputs.

$$\texttt{recognize}(P', key_{AM}, key_{FM}) \rightarrow AM, FM$$

Because the branch-based watermarker uses a blind recognizer, AM and FM can be obtained from the watermarked program by providing only the two secret keys.

Additionally, the branch-based watermarker is classified as dynamic, thus one of the secret keys, key_{AM}, is actually an input sequence to the program. By executing the program with the secret input, a trace consisting of a set of functions F is identified. The set F consists of those functions which will participate in the fingerprint calculation. During watermark recovery, if the program is executed with the input sequence, the same set of functions used for embedding will be executed. This will make it possible to identify both AM and FM.

4.1 Fingerprint Branch Function

A branch function is a special function originally proposed as part of an obfuscation technique used to disrupt static disassembly of native executables [9]. It was also used by Collberg et al. [6] in watermarking native executables, however, it was used in a manner different than in our branch-based watermark. The original obfuscation technique converted unconditional branch instructions to a call to a branch function inserted in the program. The sole purpose of a branch function is to transfer the control of execution to the instruction which was the target of the unconditional branch. Figure 1 illustrates the general idea of the branch function. To increase the versatility of the branch function we have devised an extension which makes it possible to convert conditional branches as well.

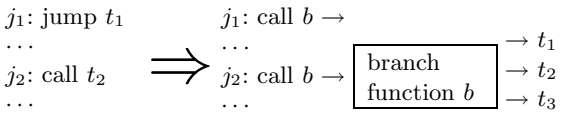

Fig. 1. Unconditional branch instructions are converted to calls to a branch function. The branch function transfers execution to the original branch target.

The FM for a program is generated as the program executes through the use of a specifically designed branch function. We call this branch function a *fingerprint branch function* (FBF). The original branch function was designed simply to transfer execution control to the branch target. In addition to the transfer of control, the FBF is also responsible for evolving a key. Each time the FBF is called a new key, k_i, is calculated. k_i is then used to aid in identifying the original branch target. The FBF performs the following tasks:

- An integrity check which produces the value v_i.
- Generation of the next function key, k_i, through the use of a one-way function, the integrity check value, and the previous key; $k_i = g(k_{i-1}, v_i)$.
- k_i is used to eventually transfer execution to the original branch target.

Within the FBF, an authorship mark and tamper detection can be incorporated. From a legal perspective, to prove ownership, it is not sufficient to simply recover a mark from a program. It is also necessary to show the watermark was intentionally embedded, i.e. recognition is not by chance. Choosing AM such that $AM = pq$ where p and q are two large primes is one possible example of a strong watermark. Since factoring is a hard problem, only the person who embedded such a watermark would be able to identify the factors p and q. To embed such an authorship mark in the FBF, the AM is encoded in the one-way function used to generate the next function key. A possible example is:

$$k_i = SHA1[(k_{i-1} \oplus AM) \parallel v_i]$$

Through the incorporation of an integrity check the FBF can detect tampering throughout the entire program. An integrity check is a section of code inserted in the program to verify the integrity of the program. The integrity checks are capable of identifying if a program has been subjected to semantics-preserving transformations or even if a debugger is present. For example, an integrity check could calculate a checksum over a block of code. If an attacker inserts breakpoints or makes some other modification to the code, the checksum will be different. We have developed a variety of different types of integrity checks. The integrity check will produce some value v_i which is then used as an additional input to the one-way function responsible for the key generation.

4.2 Embedding

The embedding of the authorship and fingerprint marks occur by injecting the FBF into the program. Selected branch instructions are then converted to calls

to the FBF. The embedding process consists of three phases. The first phase is to execute the program using the secret input sequence, key_{AM}, to obtain a trace of the program. The trace will identify the set of functions F through which execution passes. These functions will be used in the watermarking.

In the second phase of the algorithm, the branches in each function $f \in F$ are replaced by calls to the FBF. Additionally, a mapping is created between the calculated key and the replaced branch instruction. The key, branch mapping is used in phase three to construct a structure, such as a table or array, which is added to the program. The structure is accessed during execution to obtain information relating to the replaced branch instruction. This information is necessary for proper program execution. Key evolution is linked to proper program execution through the organization of the structure. Using a perfect hash function, each key is mapped to a unique location in the structure.

$$h : \{k_i, k_2, ..., k_n\} \rightarrow \{1, 2, ..., m\}, n \leq m$$

If a minimal perfect hash function is used the table size can be minimized.

Unlike the authorship mark, the fingerprint mark is not embedded in the program. Instead it is generated as the program is executed. Each function in the set F, obtained by executing the program with the secret input sequence, will produce a final function key. Each of these keys are combined in a commutative way to produce the fingerprint mark for the program. The variation in FM is obtained through the fingerprint key, key_{FM}, which is unique for each copy of the program. key_{FM} is used to begin the key evolution process in each fingerprinted function. Based on the unique key, the fingerprint for each program will evolve differently. Since the key is used to access the inserted structure, each program will contain a differently organized structure.

4.3 Recognition

As with embedding, the first step in recognizing the embedded marks is to execute the program using the secret input sequence. Execution will identify the set of functions F which have been fingerprinted, as well as the FBF itself. Once the FBF has been identified, the one-way function can be isolated to extract AM. To extract FM we have to access the location where the final function key is stored for each $f \in F$ while the program is executing. The final function keys are combined to form the FM.

4.4 Highlighted Features

The branch-based watermarking algorithm includes two features which should be highlighted. First, because the inserted structure is customized to a particular fingerprint generation, the program will only execute with the specific user key. This has the desired effect of the use of a dongle, but without the drawback of dongle distribution. In addition, the fingerprint key does not have to be stored in the program, but instead could be distributed with the program and required every time the program is executed. It is currently a concern that an attacker

could obtain the initial key and use that information in an attack. One possible solution is to leverage features of secure computing devices such as the Trusted Platform Module (TPM) available in the IBM ThinkPad laptop.

The second feature relates to the static variation between differently fingerprinted instances of a program. Because the static variation occurs only in the inserted structure, a higher level of resistance to collusive attacks can be achieved. This advantage will be further discussed in Section 6.1. Additionally, this feature can be used to enable software companies to produce and distribute fingerprinted software in the traditional manner. The program purchased would be non-functional until the user installs the software and registers it with the company. Upon registration, the user key and structure will be distributed creating a fully functioning program. Previously, if a software company wanted to tie a specific fingerprint mark to a purchaser, the user had to purchase the software directly from the company. At the time of purchase the program was fingerprinted. By using the branch-based watermark, distribution of fingerprinted software can be accomplished through pre-packaged software sold at retail stores. Installation of a fully functioning copy does require an initial Internet connection, however, this does not appear to be a drawback since most software now requires an initial registration.

5 Native Code Implementation

Our implementation of the branch-based watermarking algorithm for native code is accomplished by disassembling a statically linked binary, modifying the instructions, and then rewriting the instructions to a new executable file. The current prototype is designed to watermark Windows executable files. It provides the capability to embed an authorship mark, a fingerprint mark, and tamper detection.

As was described in Section 4.2, the embedding procedure is accomplished in three phases. In the first phase, an execution trace of the program is obtained based on the secret input sequence. Currently, identification of the set of functions used in watermarking requires manual monitoring. The program is preprocessed and a break point is inserted at the beginning of each function. As the program is executed using a debugger, information about each function encountered is recorded in a file.

During the second phase, instructions in each of the selected functions are modified. Special care must be taken in selecting which branch instructions are converted since the branch is tied to a particular key value. To ensure proper program behavior, branches are selected such that they reside on a deterministic path through the function. Without imposing this constraint, irregular key evolution will occur, resulting in the transfer of execution to an incorrect instruction. For each branch replaced, a mapping between the calculated key and the branch, target displacement is maintained.

$$\theta = \{k_1 \rightarrow d_1, k_2 \rightarrow d_2, \cdots, k_n \rightarrow d_n\}$$

θ is used in phase three to construct a table T which is stored in the data section of the binary. The table is used to store the branch, target displacement for each branch in the program which has been replaced. The first step in laying out the table is to construct a hash function such that each key maps to a unique slot in the table.

$$h = \{k_1, k_2, \cdots, k_n\} \rightarrow \{1, 2, \cdots, n\}$$

The displacements are stored in the table such that $T[h(k_i)] = d_i$.

The fingerprint branch function is a new function inserted in the program during embedding. The inserted FBF performs the following tasks:

- An integrity check which produces the value v_i.
- Generation of the next function key, k_i, through the use of a one-way function, the integrity check value, and the previous key; $k_i = g(k_{i-1}, v_i)$.
- Identification of the displacement to the next instruction via $d_i = T[h(k_i)]$, where T is a table stored in the data section and h is a hash function.
- Computation of the return location by adding the displacement d_i to the return address.

5.1 Strength Enhancing Features

Two additional features can be incorporated in the branch-based watermarking algorithm to increase the strength: integrity check branch functions and additional indirection. Each of these increases the amount of analysis required to remove the authorship and fingerprint marks.

Integrity check branch functions (ICBF) are based on the same principle as the FBF. The ICBFs are called by replaced branch instructions not used in the fingerprint generation, i.e. branches not on a deterministic path or branches in a function which is not part of the secret input. The important feature of the ICBFs is that each performs a different type of integrity check. This makes it possible to establish a check and guard system similar to that proposed by Chang and Atallah [10]. For instance the ICBFs could be used to verify that the FBF or other integrity checks have not been altered or removed.

Within the ICBFs, the integrity check value, v_i, and the branch instruction offset are used as inputs to generate a key for displacement look up. The displacements for the ICBFs are stored in the same table used by the FBF. The one-way function used to generate the key in the ICBF could be the same as that used by the FBF. If so, the authorship mark would appear in multiple locations throughout the program. If instead, different one-way functions are used, additional authorship marks could be embedded in the program, further strengthening the proof of ownership.

The second strength enhancing feature is to increase the level of indirection. Additional levels of indirection increase confusion and require more extensive analysis for an attacker. Further indirection can be incorporated in the branch-based watermarking algorithm by rerouting all calls to the ICBFs and the FBF through a single super branch function which transfers execution to the proper branch function.

6 Experimental Results

In this section we provide an evaluation of the branch-based watermarking scheme with respect to its robustness against attack and the overhead incurred. We have created a prototype implementation for watermarking Windows executable files. The current prototype only provides watermarking capabilities and does not include any of the strength enhancing features.

The evaluation was performed using the SPECint-2000 benchmark suite applications. We were unable to use *eon* and *perlbmk* because they would not build. Our experiments were run on a 1.8 GHz Pentium 4 System with 512 MB of main memory running Windows XP Professional. The programs were compiled using Microsoft's VisualStudio C++ 6.0 with optimizations disabled.

6.1 Resilience

We examined four categories of attacks to evaluate the robustness of the branch-based watermarking algorithm.

Additive Attack. In an additive attack an adversary embeds an additional watermark so as to cast doubt on the origin of the intellectual property. An attacker is successful even if the original mark remains intact, however, it is more desirable to damage the original mark. For an additive attack to be successful the program has to continue to function properly after the embedding of the second watermark. To simulate an additive attack we double watermarked the benchmark applications using the branch-based watermarking algorithm. In each case the result was an improperly functioning application. The double watermark attack fails because the integrity check detects the program alteration. A simple checksum integrity check will detect that a call to FBF1 is now a call to FBF2 or that FBF2 has been added to the program. So the attack is detected when FBF2 transfers execution control to FBF1. We believe that a similar result would be obtained if any of the currently known watermarking algorithms were used as the second watermark, however, this hypothesis is untested.

Distortive Attack. In a distortive attack, a series of semantics-preserving transformations are applied to the program in an attempt to render the watermark useless. It is the goal of the attacker to distort the software in such a way that the watermark becomes unrecoverable, yet the program's functionality and performance remain intact. To verify our hypothesis that the branch-based watermarking algorithm would be resistant to distortive attacks we subjected the benchmark applications to five different obfuscations:

1. Conversion of unconditional jumps to conditional jumps through the use of opaque predicates.
2. Conversion of unconditional jumps to calls to a branch function [9].
3. Conversion of function calls to calls to a branch function [9].
4. Basic block reordering.
5. Merging of two functions into 1 function whose control flow is regulated through opaque predicates.

In each case the resulting application was non-functional because the integrity checks detected the modification.

Collusive Attack. The most crucial attack on a fingerprinted application is the collusive attack. This occurs when an adversary obtains multiple differently fingerprinted instances of a program and is able to compare them to isolate the fingerprint. With previous watermarking algorithms, prevention of a collusive attack is often addressed through the use of code obfuscation. The general idea is to apply different sets of obfuscations to the fingerprinted programs. This will make the programs differ everywhere. This is a viable option to thwart a collusive attack, however, it may not always be feasible due to the size and/or performance overhead incurred through obfuscation.

The branch-based watermarking scheme is resistant to the collusive attack without the use of obfuscation. The only difference between two fingerprinted programs is the order of the values in the table. Thus, an attacker would have to examine the data section in order to even notice a difference.

The algorithm is still susceptible to dynamic collusive attacks, but some of those attacks can be warded off through the use of integrity checks which recognize the use of a debugger and cause the program to fail. In a dynamic attack, the only difference the adversary might notice is the value of the key that is generated at each stage, which will ultimately yield a different table slot. In order for an adversary to launch a successful collusive attack, extensive manual analysis in the form of a subtractive attack will be required to remove the fingerprint.

Subtractive Attack. In a subtractive attack, the attacker attempts to remove the watermark from the disassembled or decompiled code. If the watermark has poor transparency, an attacker may be able to discover the location of the watermark after manual or automated code inspection and then remove it from the program without destroying the software. Baring the use of a completely secure computing device, guaranteed protection against subtractive attacks is not possible. All that we can hope is that the analysis required to remove the watermark is extensive enough that an attacker finds it too costly.

The robustness against reverse engineering is partially based on the number of converted branches which contribute to the fingerprint calculation. Since the algorithm requires the branches to be on a deterministic path, the number of usable branches is being limited. During preliminary development there was question if there would be enough branches on the deterministic path to make the technique a viable option. Through analysis of a variety of different applications, we found a satisfactory number of conditional and unconditional branch instructions. Table 1 shows the total number of branches and the number of usable branches in the SPECint-2000 benchmark applications. By additionally using conditional branches we are able to significantly increase the number of usable branches. This makes the algorithm a viable option. Additionally, the data indicates that even after embedding the watermark, many branches are still available for use in the integrity check branch functions.

Table 1. Total number of branches versus the number of usable branches in the SPECint-2000 benchmark suite applications

Program	Total Branches	Usable including conditionals	Usable excluding conditionals
gzip	2843	464	170
vpr	5814	1153	674
gcc	28136	4886	3056
mcf	2028	290	89
crafty	3340	496	178
parser	5628	864	522
gap	18999	1942	1027
vortex	16144	3462	1049
bzip2	2354	457	211
twolf	4397	729	429

From our analysis, we believe that if the strength enhancing features are incorporated into the algorithm the removal of the code which generates the fingerprint will be prohibitively difficult. If the attacker is able to identify which sections of code are generating the fingerprint, he will have to manually analyze the program to identify the call instructions which are converted branch instructions. He will then have to identify the correct target instruction and replace the call with the correct branch and displacement. If the adversary only converts those branches responsible for the fingerprint generation and does not also convert the other branches, the program will fail to execute properly. This is because the integrity check branch functions are designed as a check and guard system. One of their duties is to verify that the fingerprint generating branch function has not been altered or removed. Thus, removal of the fingerprint branch function also requires removal of the integrity check branch functions. While this is not entirely impossible, the manual analysis required to accomplish such a task is extensive.

6.2 Cost

To evaluate the cost we used the SPECint-2000 benchmark suite. The overall performance of the watermarked program was evaluated using the SPEC reference inputs. The execution times reported were obtained through five runs. The highest and lowest values were discarded and the average was computed for the remaining three runs.

As can be seen in Table 2 very little performance overhead is incurred by the additional calls and integrity checks. The unwatermarked benchmark application *gcc* did not execute properly on the reference inputs so we were unable to obtain performance information suitable for comparison with the other results. However, when run using the test data no significant slowdown was observed.

The majority of the space cost incurred by the branch-based watermark is based on the size of the fingerprint branch function and the displacement table. Since the fingerprint is generated as the program executes, the size of the fingerprint does not impact the size of the watermarked program. Additionally, any difference between the converted branch and the call instruction sizes will

Table 2. Effect of watermarking on execution time

Program	Branches Used	Execution Time (sec)		
		Original (T_0)	Watermarked (T_1)	Slowdown (T_1/T_0)
gzip	79	435.52	435.52	1.00
vpr	405	479.12	480.62	1.00
mcf	24	563.07	562.55	1.00
crafty	94	326.96	326.40	1.00
parser	239	519.31	588.34	1.13
gap	742	292.20	292.01	1.00
vortex	477	316.22	316.66	1.00
bzip2	135	743.18	739.82	0.99
twolf	233	912.43	922.84	1.01

Table 3. Effect of watermarking on program size

Program	Branches Used	Program Size (KB)		
		Original (S_0)	Watermarked (S_1)	Increase (S_1/S_0)
gzip	79	100	104	1.04
vpr	405	212	252	1.19
gcc	2124	1608	2604	1.62
mcf	24	64	68	1.06
crafty	94	316	320	1.01
parser	239	184	188	1.02
gap	742	660	780	1.18
vortex	477	608	660	1.09
bzip2	135	88	96	1.09
twolf	233	316	332	1.05

contribute to the size of the watermarked application. Table 3 shows the effect watermarking had on the size of the benchmark applications. For most of the applications the size increase was minimal. *gcc* was most significantly impacted but it was also the application in which the greatest number of branches were converted. A technique to minimize the size impact is to use a minimal perfect hash function in assigning the slots in the displacement table. Our implementation did not use such a hash function, thus the results could be improved.

7 Extension to Java Bytecode

Due to restrictions placed on the Java language, a straight forward implementation of the previously described watermarking algorithm is not possible. The most limiting aspect is the difficulty in modifying the program counter register which would be analogous to the return address modification in native code. This makes it impossible to implement the branch function as it is described. However, we have devised a technique for watermarking Java applications which maintains the essence of the idea through the use of the Java interface and explicitly thrown exceptions.

The Java implementation diverges from the native code version in the second phase of the embedding. The Java FBF (JFBF) uses a completely different mechanism for transferring execution control to the branch target. The JFBF makes

use of an interface `A` which gets added to the application during embedding. Additionally, n classes `A1, A2, ..., An` are added which each implement the interface `A`. The interface `A` defines a method `branch` which is then implemented by each of the n subclasses. The main purpose of `branch` is to explicitly throw an exception. Within each of the n subclasses, `branch` will throw a different exception. Once the exception is thrown, it will be propagated up to the method which invoked JFBF. When this occurs the invoking method will find the exception in its exception table and transfer control to the instruction specified. This instruction is the target of the converted branch.

In the second phase, certain branch instructions along the deterministic path are replaced by instructions which invoke the fingerprint branch function. In the native code version we were able to replace `jmp`, `jcc`, and `call` instructions. With the Java version we are only able to replace `goto` and conditional branches. We are unable to replace `invoke` instructions because of the restrictions placed on the exception table entries. The target listed in the exception table must be an instruction within the method.

As the branch instructions are modified, two mappings are maintained. The first mapping ϕ, maps the branch target to the exception type which will be used in transferring execution control to the target instruction. The second mapping θ, maps the current key k_i to that same exception type.

$$\theta = \{k_1 \rightarrow e_1, k_2 \rightarrow e_2, ..., k_n \rightarrow e_n\}$$

ϕ is used to modify the method's exception table. For each target a new exception table entry is added. One key aspect of the Java branch-based watermarker is that for each converted branch, n entries must be added to the exception table. One of the entries is the correct target and $n-1$ are decoys. If the decoy exception entries are omitted, the branch, target pairs become obvious. Prior to execution, a Java application must pass the verification process. Verification involves checking that the class file and the bytecode meet certain constraints. Examples of the constraints include checking for consistent stack height or that local variables have been initialized. During verification, an exception edge is considered a possible execution path. Thus the targets of the decoy exceptions must be chosen such that the bytecode will still pass the Java verifier.

θ is used during phase three. In the Java version, an array is used to store objects instead of a displacement table. The array T stores objects which are subclasses of A, so a combination of objects `A1, A2, ..., An`. The array is constructed again using a hash function which uniquely maps each key to a slot in the array. The objects are stored such that $T[h(k_i)] = A_j$, where A_j's `branch` method throws the exception e_i.

To regulate execution control and generate the fingerprint the JFBF performs the following tasks:

- An integrity check producing v_i.
- Generation of the next method key, k_i, through the use of a one-way function; $k_i = g(k_{i-1}, v_i)$.

- Object look up through the use of an array, the key, and a hash function, `A a` $= T[h(k_i)]$.
- Call the method `branch` using the object `a`, `a.branch()`.

Currently, we only have a preliminary implementation of the branch-based software watermarking algorithm for Java bytecode. Because the implementation only includes minimal functionality we have yet to perform a thorough experimental evaluation. As our future work we plan to carry out such an evaluation.

8 Conclusion

In this paper we described a novel approach to software watermarking, branch-based watermarking, which incorporates ideas from code obfuscation and tamper detection to increase robustness against determined attempts at discovery and removal. Our technique simultaneously provides proof of ownership and the capability to trace the source of the illegal redistribution. This is an improvement over previous techniques which required the developer to choose between embedding an authorship mark or a fingerprint mark. Additionally, the branch-based watermarker provides a solution for distributing pre-packaged, fingerprinted software which is uniquely linked to the purchaser.

The branch-based watermark prototype demonstrates that the technique can successfully thwart both additive and distortive attacks. The technique also demonstrates a higher level of resistance to subtractive and collusive attacks. Previous fingerprinting techniques addressed the prevention of collusive attacks through the use of code obfuscation which introduces additional overhead. The only static variation introduced by the branch-based watermark is in the table. This makes it more highly resilient to collusive attacks even without the use of obfuscation. Additionally, the overhead associated with the technique is quite minimal and should be tolerable for most applications. By eliminating automated attacks, such as semantics-preserving transformations, and many of the common manual attacks, attackers are forced to use more complex and costly techniques. Thus, attackers who lack the necessary skill or find the required attacks to be too expensive will be eliminated.

References

1. International Planning and Research Corporation: Sixth annual BSA global software piracy study (2001)
2. Venkatesan, R., Vazirani, V., Sinha, S.: A graph theoretic approach to software watermarking. In: Information Hiding Workshop. (2001)
3. Stern, J.P., Hachez, G., Koeune, F., Quisquater, J.J.: Robust object watermarking: Application to code. In: Information Hiding Workshop. (1999)
4. Qu, G., Potkonjak, M.: Hiding signatures in graph coloring solutions. In: Information Hiding Workshop. (1999)
5. Collberg, C., Thomborson, C.: Software watermarking: Models and dynamic embeddings. In: Symposium on Principles of Programming Languages. (1999)

6. Collberg, C., Carter, E., Debray, S., Huntwork, A., Kececioglu, J., Linn, C., Stepp, M.: Dynamic path-based software watermarking. In: Conference on Programming Language Design and Implementation. (2004)
7. Nagra, J., Thomborson, C.: Threading software watermarks. In: Information Hiding Workshop. (2004)
8. Cousot, P., Cousot, R.: An abstract interpretation-based framework for software watermarking. In: Symposium on Principles of Programming Languages. (2004)
9. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: ACM Conference on Computer and Communications Security. (2003)
10. Chang, H., Atallah, M.J.: Protecting software code by guards. In: ACM DRM workshop. (2001)