SWuS : Software Watermark using Slices

Amarnath Mullick   Y.N. Srikant

IISc-CSA-TR-2004-7
http://archive.csa.iisc.ernet.in/TR/2004/7/

Computer Science and Automation
Indian Institute of Science, India

September 2004

SWuS : Software Watermark using Slices

Amarnath Mullick Y.N. Srikant*

Abstract

Like all other forms of *intellectual property*, software is prone to illegal copying. *Software watermarking* is the process of embedding some information in a program in a way that it cannot be easily removed by an adversary. However, it can be recovered at a later time with the help of a recognizer to assert the ownership rights of the original software author. A spurt of research activity has taken place in this field recently and a number of software watermarking algorithms proposed. However, almost all of them have one or the other weakness either due to their lack of stealth or robustness or due to their adverse effect on the efficiency of the watermarked program.

In this report we develop a novel software watermarking method, SWuS, based on the concept of program slice which is extremely *stealthy* and *robust* and does not affect the performance of the subject program by any significant factor. Also we propose how to use existing error correcting codes to add redundancy to the embedded watermark and thus add to its *resilience*.

1 Introduction

1.1 Motivation

A lot of research has been done to build security systems to prevent unauthorized access to a machine and protect the data residing in it. Software is also vulnerable to theft and misuse as any other form of data, and hence needs to be protected. *Software piracy* has been a major threat to the software industry. In a recent study done by the Business Software Alliance it was found that the global economic impact of pirated software totaled nearly US $13 billion in 2002 [BSA03].

*Author for TR correspondence. `srikant@csa.iisc.ernet.in`

Given the enormous investment in time, money and intellectual capital in software development such *intellectual property rights violation* in the form of software piracy is a matter of great concern and needs to be countered. One radical solution to stop illicit use of software is to provide the *services* of the software rather than selling it. This is the *application server* model, where a user never gets physical access to the application. Instead, he connects to a server to run the program remotely, paying a small amount for each access. But this approach has inherent performance and scalability problems.

Another sure way to prevent software modification or copying is to prevent an adversary from seeing the code at all. Some researchers have proposed *encrypting* the code of an application. Unfortunately, this works only if the entire decryption/execution occurs at the hardware level. Otherwise, the decrypted code can always be intercepted defeating the whole cause for encryption. Additionally, the runtime system must decrypt the code before execution every time, which can degrade performance.

One intriguing defense against software piracy can be *software aging* [JR00] where frequent updates and new features are provided to the users that lessen the utility of the older versions of software. The pirates are forced to provide their own updates, thereby taxing their resources and increasing the likelihood of detection. However, this approach has only limited scope since the end user often doesn't need to have the latest version of the software.

The *hardware assisted* scheme of protection prevents programs from running unless a specific floppy, CD, dongle, or secure digital memory card is present. To enable full functionality the program must retrieve information stored on the hardware component that validates use of program on the machine. To defeat such a mode of protection the attacker only needs to identify and disable that part of the software which retrieves and checks information from the hardware component. In addition, all hardware based protection mechanisms suffer from the general drawback of lacking user-friendliness.

There is a general consensus regarding the impossibility of complete protection of client code [BGI+01]. In such a scenario where it is not always possible to protect software from being copied, it is extremely important for the owner to actually demonstrate his ownership of intellectual rights in a convincing way, in case of software theft.

Watermarking, a technique widely used to authenticate objects in digital media and hardware designs, can be used likewise to embed ownership marks in software. *Software watermarking* involves altering the given program to embed some form of *ownership information* into the program. The mark should be recoverable from an *allegedly* illegal copy of the software by a watermark recognizer and its existence will prove the ownership claim of the author. However, the problem of software watermarking is not so much in the embedding of the watermark, as it is in making sure the watermark survives deliberate attempts to remove it from a program by an adversary through different program transformations.

1.2 Relevant Work

Different software watermarking schemes have been proposed in the literature. These schemes can be broadly classified into two categories, static and dynamic.

1.2.1 Static Watermark

Static watermarks are stored in the body of the application itself and do not need the program to be executed for its recognition. They can be either *static code watermarks* or *static data watermarks* depending on which part of the application the watermark is stored.

Moskowitz [MC96] proposes a data watermarking method where the watermark is encoded in a digital media like video or audio using the existing media watermarking algorithms and then storing that in the data section of the program.

Code watermark can be formed by encoding a bit in the relative ordering of two statements S_1 and S_2, in the program, having no data or control dependencies between them. A similar approach is taken by Davidson and Myhrvold [DM96] where they embed the watermark in the sequence of the basic blocks in the control flow graph of the program. Venkatesan et al. [VVS01] use a graph theoretic approach to embed the watermark in the control flow structure of the program and Stern et al. [SHKQ99] use a spread spectrum technique to embed the watermark in the relative frequencies of different types of instructions in the program.

In another approach to code watermarking, Arboit [Arb02] presents a method to watermark Java programs using *opaque predicates* (predicates whose truth value is known at watermarking time).

Static watermarks are extremely vulnerable to distortive attacks by obfuscation or compiler optimizations. Even simple automated attacks which do not affect program performance by any significant factor may remove them. Tamper-proofing may help in some cases, but that removes the stealthiness of the watermark.

1.2.2 Dynamic Watermark

Dynamic watermarks, on the other hand, are stored in the execution state of the program. So to recover the mark the program needs to be executed, usually with a predetermined special input sequence. The dynamic watermarks differ in which part of the program state the watermark is stored and in the way it is extracted. In [CT99], Collberg et al. enumerates three important dynamic watermarking techniques:

Easter Egg watermark gets activated for a highly unusual input to the application and produces some action that is immediately perceptible by the user. Easter Egg watermarks are not resilient. There are even web repositories of such watermarks [EEG].

Dynamic data structure watermark stores the mark in some data structure of the application after it is run with the watermarking input. The watermark has to be extracted from some of the program variables and can be done either by a dedicated watermark extraction routine which is linked with the executing program or by running the program under a debugger. These watermarks are more resilient than Easter Eggs, but still extremely susceptible to obfuscating attacks.

Dynamic execution trace watermark embeds the watermark within the trace (either instruction sequence or addresses accessed, or both) of the program when it is run with the watermarking input and extracted from certain, possibly statistical, properties of the executed instruction sequence or address trace. Optimizing transformations can obliterate the requisite properties required to extract the watermark.

In [CT99] Collberg and Thomborson also proposes a special type of dynamic data structure watermark where the watermark is embedded in the topology of a dynamically built graph structure. Due to the inherent difficulty of analyzing code that manipulates dynamic graph structures it is difficult to find transformations that would change the graph and yet ensure that the program behavior remains unchanged for an attacker who does not know which part of the code actually builds the watermark graph. In [TNSH] a method called *constant encoding* is introduced to tamper-proof the software watermark. It is based on transforming constant values in the text of the watermarked program into function calls whose value depends on the watermark graph.

However, though this type of watermark may withstand many distortive attacks, it is extremely unsuitable for a large class of programs where the code inserted for building the graph won't mix well with the original program (e.g. numerical programs) and hence this code can be easily located and removed. Also, if the graph building code is executed frequently, which would be the case if the above mentioned tamper-proofing scheme is used, it may affect the performance of the program. On the contrary, if the code is made to execute only on the watermarking input, statistical analysis of the program execution will locate it easily.

Another dynamic watermarking scheme is proposed by Curran et al. [CHC03] using spread spectrum analysis, by extracting a vector from the call graph depth at distinct points during the execution of the program on some particular input. This can be attacked by adding spurious function calls in the program, thereby changing the call graph depth information extracted.

A new class of software watermarking is proposed by Cousot and Cousot [CC04] called *abstract software watermarking*. The key idea here is to add some code (stegomark) to the program, and the static analysis of the program under the framework of abstract interpretation would reveal the watermark. The scheme has been exemplified by an instance based on modular constant propagation. This scheme also suffers from the loose coupling of the stegomark with the original program. Also certain obfuscation transformations can confuse the recognizer from recognizing the watermark variable. For example, adding new path to the program protected by false opaque predicates, that would assign some arbitrary value to the watermark variable, will confuse the recognizer trying to find the watermark analyzing the constant propagation.

Collberg et al., in [CCD+04], proposes a dynamic path based approach to watermarking where the basic idea is to embed the mark in the branching structure of the program. They also introduce error correcting and tamper proofing mechanism to add to the resilience of their scheme.

1.3 Contributions

We develop a novel software watermarking method, SWuS, based on the concept of program slice which is extremely *stealthy* and *robust* and does not affect the performance of the subject program by any significant factor. Also we propose how to use existing error correcting codes to add redundancy to the embedded watermark and thus add to its *resilience*.

1.4 Outline of the Report

The rest of the report is organized as follows. Section 2 formalizes the notion of software watermarking. In Section 3 we introduce a novel approach to watermarking, where the watermark is embedded in an unreachable state of a program slice. Encoding the watermark value to add redundancy is proposed in Section 4. Section 5 discusses the robustness of the proposed watermarking algorithm under different attacks and also suggests enhancements to add to its resilience. The various properties of the scheme is discussed in Section 6. Section 7 looks at the functional requirements of an automated watermarking scheme satisfied by the slice based watermarking. The overview of the implementation of an automated tool based on this scheme is given in Section 8. Finally, Section 9 concludes the report.

2 The Problem of Software Watermarking

Before we discuss our proposed scheme, we formalize the problem of software watermark embedding and its recognition.

Let $\mathcal{P}$ be the set of programs and $\mathcal{W}$ be the set of watermarks. The *watermark embedding function* is

$$\mathcal{E} : (\mathcal{P} \times \mathcal{W} \times \mathcal{A}_{\mathcal{E}}) \to \mathcal{P}$$

where $\mathcal{A}_{\mathcal{E}}$ is the set of auxiliary information fed to $\mathcal{E}$. The *watermark recognition function* is

$$\mathcal{R} : (\mathcal{P} \times \mathcal{A}_{\mathcal{R}}) \to \mathcal{W}$$

where $\mathcal{A}_{\mathcal{R}}$ is the set of auxiliary information that may be provided to the recognizer $\mathcal{R}$. The watermark embedding function $\mathcal{E}$ and the corresponding recognition function $\mathcal{R}$ together forms a watermarking method $\mathcal{M} = (\mathcal{E}, \mathcal{R})$.

The embedded mark $w \in \mathcal{W}$ should have some mathematical property which should ascertain that its presence in the program is not by accident, but deliberate. For example, w may be a large number $\mathcal{N} = \mathcal{N}_1 \times \mathcal{N}_2$, where $\mathcal{N}_1$ and $\mathcal{N}_2$ are large primes.

Correctness

Given a program $p \in \mathcal{P}, w \in \mathcal{W}$ and some auxiliary information $A_e \in \mathcal{A}_{\mathcal{E}}$,

$$\mathcal{E}\,(p, w, A_e) = p_w$$

where p_w is the watermarked program, for *correctness* of $\mathcal{M} = (\mathcal{E}, \mathcal{R})$,

$$\mathcal{R}(p_w, A_r) = w, \quad A_r \in \mathcal{A}_{\mathcal{R}}$$

The tuple (A_e, A_r) is referred to as the *watermark key* and generally kept secret. Optionally, $\mathcal{E}$ and $\mathcal{R}$ can be made public.

Attacks and Robustness

The problem of watermarking is not as much in embedding the watermark as it is in being able to identify it in a program that has undergone one or more transformations made by adversaries, who want to render the watermark useless. A matter of prime interest is whether the recognizer $\mathcal{R}$ would be effective when the watermarked program has undergone a transformation $t : \mathcal{P} \to \mathcal{P}$.

The watermarking method $\mathcal{M} = (\mathcal{E}, \mathcal{R})$ is *robust* over the set $T \subset \mathcal{T}$, $\mathcal{T}$ being the set of program transformations, if

$$\forall t \in T : \mathcal{R}\left(t\left(p_w\right), A_r\right) = \mathcal{R}(p_w, A_r) = w$$

False Recognition

Ideally, there should be no *false recognition* by the recognizer $\mathcal{R}$, that is,

$$\begin{gathered} \mathcal{R}(p', A_r) \neq w \\ \text{if } p' \notin \left\{t\left(\mathcal{E}\left(p, w, A_e\right)\right) : \forall t \in T\right\} \end{gathered}$$

Note that the set T includes the *identity* transformation.

3 SWuS: Software Watermark using Slices

We introduce SWuS, a novel method of software watermarking which uses the concept of program slice. The key idea in SWuS is to hide the watermark in a state of the slice which is unreachable during normal program execution and is unmasked by the recognizer during watermark extraction.

3.1 Program Slicing

Program Slicing was introduced by Weiser [Wei84] as a method used by experienced computer programmers for abstracting from programs. Slicing reduces a program to a minimal form which retains a desired subset of the program behavior *(slicing constraints)*. For example, given a set of variables V and a program point P, the set of program statements that directly or indirectly contribute to the values assumed by the variables in V at P, constitute a *slice* S. S is an independent program which, when executed, will result in the variables getting the same values as they would have for the original program at P given similar input. Thus the slice would faithfully represent the original program in the domain of the specified subset of program behavior. Figure 1 shows a C function and the corresponding slice on the value of the variable `max` at the statement `L`.

3.2 Overview of SWuS

A slice with respect to a variable `x` and program point W is extracted from the subject program p, processed by the watermark embedder $\mathcal{E}$ and then reintroduced in p to obtain the watermarked program p_w. The modification done is such that p_w retains its resemblance to p in terms of input output relations as also other observable behaviors, like execution time, memory requirement, etc. Apart from the processing, $\mathcal{E}$ also collects some information which will be later passed on to the recognizer $\mathcal{R}$ as auxiliary information A_r.

Aided by the information A_r as also hints left by the processing done by $\mathcal{E}$ in the watermarked program, the recognizer $\mathcal{R}$ is able to locate the *slicing constraints*, i.e., the variable `x` and the program point W, even in a possibly transformed version of p_w. Next, the recognizer finds the slice with respect to these constraints, referred to as the *watermark slice* henceforth. Next, $\mathcal{R}$ makes some trivial and predefined changes to the watermark slice unveiling the hitherto unreachable state in which the watermark has been hidden. When this slice is executed with any one of a set of given input sequences, called the *watermarking inputs*, the watermark is constructed and gets assigned to the variable `x`.

Thus SWuS has three phases:

Watermark Embedding This is the stage where the program slice is extracted, modified as required for watermarking and reintroduced in the program.

Watermark Recognition Here the watermark slice is recognized and extracted from the *possibly transformed* watermarked program. The slice is then made to undergo some trivial and predefined set of changes.

Watermark Construction Here the resultant slice is executed with one of a given set of the watermarking inputs to obtain the watermark.

It is to be noted that the proposed scheme is that of a *blind watermark* in that the recognizer $\mathcal{R}$ does not need the original copy of the watermarked program, but only the possibly transformed watermarked program and some auxiliary information A_r.

```
int max, min, a[100];
max = INT_MIN;
min = INT_MAX;
scanf("%d", &n);
for(i = 0; i < n; i ++)
{
    scanf ("%d", &a[i]);
    if (a[i] > max)
L:    max = a[i];
    if (a[i] < min)
      min = a[i];
}
printf("min = %d, max = %d\n",
          min, max);
```

Program fragment

```
int max, a[100];
max = INT_MIN;
scanf ("%d", &n);
for (i = 0; i < n; i ++)
{
    scanf ("%d", &a[i]);
    if (a[i] > max)
L:    max = a[i];
}
```

Program slice

Figure 1: Example of Program Slice

3.3 Definitions

Let us introduce some terms to be used frequently in the discussion in the following sections:

Subject Program This is the program under consideration to be watermarked.

Fragment Variable This is one of the subject program variables which will contain a fragment of the watermark.

Slice Assignment This is a subject program statement where the fragment variable is assigned the value of some expression.

Guard Predicate This is a predicate (e.g., an *if* statement) in the subject program which must evaluate to true for the *slice assignment* statement to be executed.

Guard Tag This is some form of tag added to the *guard predicate* for the recognizer to be able to find it. The *guard tag*, which is generally some code fragment, should not affect program behavior.

Watermark Predicate The guard predicate marked by the guard tag in the watermarked program is the watermark predicate.

Watermark Code The code that replaces the slice assignment in the watermarked program. Parts of it may be made into function calls, and the corresponding functions are the *watermark functions*.

Watermark Assignment The watermark code ends with an assignment statement. This is the watermark assignment.

Watermark Cover Code The *watermark predicate* and the *watermark code* including the *watermark functions*, as also any other modifications made to the program due to watermarking, e.g., variable declarations added, constitute the *watermark cover code*, or simply the *cover code*.

3.4 Embedding the Watermark

Following are the steps for embedding the watermark fragment in the program p:

1. The embedder chooses the fragment variable, the slice assignment and its guard predicate. It should make sure that the slice assignment and the guard predicate are chosen such that the former is executed in p if and only if the latter evaluates to `true`.

2. It adds the guard tag to the guard predicate forming the watermark predicate. One example for the tag is an opaque predicate [CTL88].

3. It analyzes the slice assignment and accordingly manufactures the *watermark code*. The structure of this code is discussed in Section 3.5

4. The slice assignment is replaced with the watermark code.

3.5 Watermark Cover Code

As mentioned in Section 3.2, the watermark cover code that is introduced by the watermark embedder $\mathcal{E}$ is so designed that the observable program behavior is unchanged, but the watermark is embedded in an unreachable state of the watermark slice that would later be extracted by the recognizer $\mathcal{R}$. In this section we will discuss the structure and properties of the cover code in details.

Consider the pseudo code in Figure 2. Let it be a part of the subject program p. Statement `S` is chosen as the slice assignment where the expression `E` is being assigned to the fragment variable `x`. `GP` is the guard predicate of the slice assignment.

```
if (GP)     /* guard predicate  */
{
    ...
S:  x = E;  /* slice assignment */
    ...
}
```

Figure 2: Code Fragment of Subject program

```
if (GP op GT)  /*watermark predicate*/
{
    ...
S1: x = wm1();
S2: x += wm2();/*watermark assignment*/
    ...
}
```

Figure 3: Code fragment after Watermark Embedding

The watermark embedder changes the above code to the form shown in Figure 3, in the watermarked program p_w. `GT` is an opaque predicate used as the guard tag. `op` is either `AND` or `OR` depending on whether the opaque predicate is always true or always false, respectively. The tagged guard predicate, that is, `(GP op GT)` forms the watermark predicate. The slice assignment statement `S` in p is replaced by the watermark code comprising of the statements `S1` and `S2` in p_w.

`S2` is the watermark assignment as defined in Section 3.3. It is this statement that the recognizer must find out later for watermark extraction. The watermark will be assigned to the fragment variable `x` during watermark construction in this watermark assignment statement.

The bulk of the watermark code is contained in the *watermark functions* `wm1` and `wm2`. The pseudo code of the two functions is shown in the Figure 4.

```
wm1()
{
    if(P1)
            A1;
    else if(P2)
            A2;
    :
    else if(Pn)
            An;
}

wm2()
{
    if(Q1)
            B1;
    else if(Q2)
            B2;
    :
    else
            Bn;
}
```

Figure 4: The watermark functions

The predicates $P = \{P_1, P_2, \ldots, P_n\}$ and $Q = \{Q_1, Q_2, \ldots, Q_{n-1}\}$ satisfy the following properties:

1. $\forall P_i \in P : P_i \Rightarrow GP$, that is, if any predicate in P is true, then the guard predicate, GP, is true.

2. $GP \Rightarrow \bigvee_i P_i$, that is, if GP is true, then at least one of P_i is true. This can be ensured trivially by making P_n same as GP.

3. $\forall k \in [1, n-1] : P_k \Leftrightarrow Q_k$

It is to be noted that in the pseudo code shown in Figure 4, though there is the predicate P_n, there is no corresponding predicate Q_n.

$A = \{A_1, \ldots, A_n\}$ and $B = \{B_1, \ldots, B_n\}$ are set of program statements or *actions*, that are executed depending on which predicate(s) in P and Q evaluate to true during program execution.

The properties enumerated above, along with the structure of the the two functions `wm1` and `wm2`, imply that for a given instance of program execution, if A_l is executed, then so is B_l, for some $l \in [1, n]$, and no other action in A or B is executed. Additionally, the actions in A and B are so designed that the effect of execution of the watermark code (i.e. statements `S1` and `S2`, refer Figure 3) in p_w is equivalent to the execution of the slice assignment statement `S` in the original program p. This ensures that irrespective of which pair of actions (A_l, B_l) are executed, the program behavior remains unaffected for all possible inputs.

B_n is a *specially designed* action and has the following properties:

1. When B_n is executed after A_n has been executed, the effect of the watermark code in p_w is same as the slice assignment `S` of p, as discussed earlier.

2. When B_n is executed alone in the watermark code, the watermark value gets assigned to the fragment variable `x` in the watermark assignment statement `S2`. Though there is no such possibility during normal execution of p_w, it is this unreachable state that would later be exploited by the recognizer $\mathcal{R}$.

3.6 Watermark Recognition

Recognition is the most critical part of any watermarking algorithm. We will base our discussion about watermark recognition on a *possibly transformed* version of p_w. In this section we will mention the basic tasks to be performed by the recognizer and also show how at all the watermark is constructed.

Later on, in Section 5 we will elaborate on the recognition mechanism under different possible attacks.

The recognizer $\mathcal{R}$ basically needs to identify the following from the code under inspection:

1. The watermark predicate.
2. The fragment variable.
3. The watermark assignment.

The auxiliary information A_r that can be fed to $\mathcal{R}$ are:

1. The location of the watermark predicate in the subject program.
2. The guard tag added to the guard predicate.
3. The name of the fragment variable.
4. The location of the watermark assignment in the subject program.

If p_w is unchanged from its form after watermarking then the work of the recognizer having the above information A_r is trivial. Once the recognizer has identified the fragment variable and the watermark assignment, it finds the watermark slice with respect to these, which would obviously include the guard predicate too. $\mathcal{R}$ then modifies the watermark predicate such that it always evaluates to true. This can be done trivially by `OR`'ing it with `true`. An implication of this is now when the slice is executed with certain inputs, called the *watermark inputs*, program flow enters the block guarded by the watermark predicate and subsequently the watermark code is executed even though the original guard predicate `GP` is false. Now,

$$\begin{array}{rl} \text{Since, } P_i \Rightarrow GP & \forall i \in [1,n] \\ \text{Therefore, } \neg GP \Rightarrow \neg P_i & \\ \text{Also as, } P_k \Leftrightarrow Q_k & \forall k \in [1,n-1] \\ \text{Therefore, } \neg GP \Rightarrow \neg Q_j & \forall j \in [1,n-1] \end{array}$$

Hence, when program flow comes to the watermark code in spite of `GP` being false, none of the predicates in P and Q will evaluate to true. Consequently, the only action to be executed here is B_n. Due to property of B_n discussed earlier, the watermark value gets assigned to the fragment variable in the watermark assignment.

Note that we need not restrict ourselves to two watermark functions, as shown in Section 3.5, as long as the basic principle of designing the watermark code is adhered to. The code can even be inlined without any function calls at all.

3.7 Example

Let us illustrate the watermark embedding and recognition with an example C program fragment shown in Figure 5.

```
int foo(int a, int b)
{
    int x;
    ...
    if(a < b)
S:    x = a + b;
    else
      x = a - b;
    ...
}
```

Figure 5: Example subject program (p) fragment

`x` is the fragment variable and the statement labeled `S` is the slice assignment. The predicate `(a < b)` in the `if` statement is chosen as the guard predicate.

Figure 6 shows the code after it has been processed by the embedder $\mathcal{E}$. The opaque predicate `(7*a*a-1 == b*b)` serves as the guard tag and the modified `if` statement `SP` is the watermark predicate. The operator `OR` has been used to add the guard tag to the guard predicate, as the opaque predicate being used for the tag always evaluates to false, irrespective of the values of `a` and `b`. The statements `S1`, `S2` and `S3` comprise the watermark code and `S3` is the watermark assignment statement.

The body of the watermark functions `wm1` and `wm2` are shown in the Figure 7. The predicates P_1, P_2 and Q_1 as well as the actions A_1, A_2, B_1 and B_2 satisfy the properties enlisted in Section 3.5, under the assumption that `a` and `b` are always positive, an information which the programmer can provide. Also the watermark value embedded is 85.

During recognition, $\mathcal{R}$ finds the watermark assignment `S3` and the watermark predicate `SP` and modifies the code as shown in Figure 8. Note the change made to statement `SP` where the watermark predicate is `OR`ed with 1. Now, when this is run with a watermarking input, execution will come to the watermark code even though the guard predicate is false, that is, `a >= b`. All of P_1, P_2 and Q_1 will be false and only the action B_2 will be executed resulting in the fragment variable `x` being assigned the watermark value 85.

```
int foo(int a, int b)
{
    int x, temp;
    ...
SP: if(a < b || 7*a*a-1 == b*b)
    {
S1:   temp = a + b;
S2:   x = wm1 (temp, a, b);
S3:   x += wm2 (x, temp, 2*a, b);
    }
    else
      x = a - b;
    ...
}
```

Figure 6: Example watermarked program p_w

```
int wm1(int t, int a, int b)
{
  if(2*a < b)      /* Predicate P1 */
    return t/2;    /* Action A1    */
  else if (a < b)  /* Predicate P2 */
    return t - 85; /* Action A2    */
  else
    return 0;
}

int wm2(int x, int t, int aa, int b)
{
  if(aa < b)       /* Predicate Q1 */
    return x + t%2;/* Action B1    */
  else
    return 85;     /* Action B2    */
}
```

Figure 7: Example watermark functions

```
int foo(int a, int b)
{
    int x, temp;
    ...
SP: if((a < b || 7*a*a-1 == b*b) || 1)
           /* Note the change above */
    {
S1:   temp = a + b;
S2:   x = wm1(temp, a, b);
S3:   x += wm2(x, temp, 2*a, b);
    }
    else
      x = a - b;
    ...
}
```

Figure 8: After Watermark Recognition

4 Encoding the Watermark for Redundancy

In the following section (5) we will discuss the various measures that can be taken to make the watermark recognition mechanism discussed in Section 3.6 resilient to different attacks. However, it is to be noted that no watermarking method can be *completely* full-proof. So, assuming that in spite of our best efforts, an adversary still can remove the watermark or at least distort it, we propose to add redundancy to the watermark embedded in the program. We encode the watermark value with some redundancy, split it into parts and embed each part separately as discussed in Section 3.4 into different parts of the program p.

This gives three very important advantages to the watermarking scheme:

1. Even if an attack removes some of the embedded watermark fragments in p_w and/or tampers a few others, still up to an acceptable limit, the watermark can be reconstructed from the remaining parts.

2. Depending on the degree of redundancy the watermark can be recovered even if only certain portion of the watermarked program is available.

3. The scheme of encoding the watermark and then splitting gives a lot of flexibility in choosing the size of the watermark value.

We propose to use the *Reed-Solomon code* [RS60] which is a block based error correcting code having a wide range of applications in digital communications and storage. It is based on *Galois or finite field* arithmetic. A Reed-Solomon codeword is generated using a special polynomial called the *generator polynomial*. All valid *codewords* are exactly divisible by the generator polynomial. The general form of the generator polynomial is

$$g(x) = (x - \alpha^i)(x - \alpha^{i+1}) \ldots (x - \alpha^{i+2t})$$

where α is a *primitive element* in the field, and $2t$ is the number of *parity* blocks to be added in the encoding. The codeword is constructed using:

$$c(x) = g(x).i(x)$$

where $i(x)$ is the data and $c(x)$ is the codeword.

A Reed-Solomon code is specified as $RS(n, k)$ with s bit symbols which means the encoder takes k data symbols of s bits each and adds parity symbols to make an $n = 2^s - 1$ symbol codeword. The parity symbols, numbering $2t = n - k$ are also of s bits each. The Reed-Solomon decoder can correct e erased symbols and r erroneous symbols as long as $e + 2r \leq 2t$. If the number of erasures and errors exceed the tolerance limit either the decoder will detect that it cannot recover the original codeword and indicate this or will mis-decode and recover an incorrect code.

A popular Reed-Solomon code is RS(255,223) with 8 bit symbols. Each codeword contains 255 bytes, of which 223 bytes are data and 32 bytes are parity. However, for our watermarking purpose we generally need the data to be around 8 bytes. This can be done using RS(255, 223) code, where 215 of the 223 data bytes are kept as zero. After encoding, we have 40 bytes (8 data bytes and 32 parity bytes) of information to be embedded as watermark. These 40 bytes can be split as 40 single byte watermark fragments, and embed in different parts of the program. During watermark recognition, out of the 40 fragments if r fragments are recovered incorrectly and e fragments are not at all recovered, still the watermark can be reconstructed as long as $e + 2r \leq 32$.

It is to be noted, this is only one example of how the Reed-Solomon code can be used in SWuS. The parameters can be changed as and when required depending on the subject program. For example, if the program is not sufficiently large to embed as many as 40 watermark fragments, we can embed 20 two-byte fragments instead of the 40 single bytes or else use a different Reed-Solomon code to cut down on the number of watermark fragments. Conversely, if the subject program is large, more redundancy in the form of larger number of parity symbols can be added.

5 Refinements to Counter Attacks

As mentioned earlier, the recognizer needs to identify the watermark predicate, fragment variable and the watermark assignment to extract the watermark. Several transformations are possible on the watermarked program, which may make one or more of the information in A_r listed in Section 3.6, invalid or unusable. For the watermark recognition to be of any practical use it should be able to extract the watermark in spite of such transformation.

Manual attacks, in which the attacker invests a lot of time and resources to actually understand the full semantics of the program code, can always foil a watermarking scheme. However, we are more concerned with *automatic* attacks on a watermarked program.

As mentioned in Section 2, given

$$\mathcal{E}\,(p, w, A_e) = p_w$$

the watermarking method $\mathcal{M} = (\mathcal{E}, \mathcal{R})$ is *robust* over the set $T \subset \mathcal{T}$, $\mathcal{T}$ being the set of program transformations, if

$$\forall t \in T : \mathcal{R}\,(t\,(p_w), A_r) = \mathcal{R}(p_w, A_r) = w, A_r \in \mathcal{A}_{\mathcal{R}}$$

We are mainly concerned with $T_s \subset \mathcal{T}$, the set of automated *semantics-preserving* transformations, where,

$$\text{if } p_s = t_s(p), \quad t_s \in T_s,$$

p_s preserves the *input-output behavior* of p. It is highly desirable that a watermarking scheme is robust over the set of transformations T_s, e.g., *translation*, *optimization* and *obfuscation*.

In the following sections we will look in details into the various attacks in the form of the above class of transformations, and propose methods to counter the same. It is to be noted that for an attack to be considered as successful it should not only defeat the recognizer, but the resultant program after the attack should preserve enough of the *essence* of the original, to be of some *value* to the attacker.

5.1 Subtractive Attacks

In a subtractive attack the adversary tries to detect the location of the program where the watermark has been embedded and attempts to crop it out from the program, at least to the extent, when the mark is no longer recognizable.

5.1.1 Dead Code Elimination

If the watermark is hidden in dead code, e.g., in [MIM+04], static analysis easily finds it out and eliminates it. In SWuS, the added watermark code is added as the replacement of an existing program statement and hence is an integral part of normal program execution. So subtractive attacks in the form of dead code elimination fails.

5.1.2 Statistical Analysis

Watermarking schemes banking solely on opaque predicates, as in [Arb02], are susceptible to attacks based on statistical analysis of program execution. Since opaque predicates always evaluate to either true or false, the code that is executed depending on their truth value is either always or never executed, respectively. An attacker can then concentrate on that part of the program showing such behavior for finding the watermark.

Watermark schemes depending on some special and generally unusual input for which the watermark is built (e.g., [CT99]) are also not resistant to this type of attacks. The program execution is monitored for a large collection of possible inputs. Here the watermark cover code can be characterized as a part of the very infrequently executed portions of the program, thus risking the chance of detection.

Such statistical analysis of the frequency of execution of some program fragment can't help in locating the cover code in SWuS. The frequency would be same as that of the slice assignment which the watermark code replaces and hence doesn't have any specific statistical property that an attacker can exploit.

Additionally, the predicates in P and Q should be such that all the actions in A and B are executed for one or the other input. To ensure this,

$$P_{i_1} \not\Rightarrow P_{i_2} \text{ for any } i_1, i_2 \in [1, n] \text{ and } i_1 > i_2.$$

This property of the predicates in the watermark functions shown in Figure 4 implies that none of the predicates in an upper `if` statement masks another below, so that all the actions get their chance to be executed for some input.

5.1.3 Literal Constants

Large literal constants are generally unusual in real programs. Use of such constants may increase the visibility of the cover code. An attacker can concentrate on parts of the program using such constants to locate and crop off the watermark code.

Since we are using Reed-Solomon to encode the watermark value and then splitting it into parts (each of length 1 byte), the literal constants used are never more than 255. In fact, use of literal constants in the watermark code as we did in the example in Figure 7, can be avoided altogether. Instead of using any of a set of watermark inputs, we first decide on a specific watermark input I_w to be used during recognition. Now, using the variables in scope at the watermark code, we make up an expression which will evaluate to the watermark value when the program is executed with the input I_w. For example, say our watermark fragment value is 85, and we know there are two variables `c` and `d` in scope which will take the value 2 and 3, respectively for the input I_w. So the instances of 85 in the watermark code can be replaced with the expression `(b<<5) + 7*c`.

5.1.4 Intrinsic Security from Subtractive Attack

The main principle of SWuS requires that the watermark code replaces a statement (the slice assignment) in the original program. Hence, even if a subtractive attack is able to locate the code and removes it without inserting proper compensation code, the original program behavior will change. Hence, SWuS is intrinsically secure from subtractive attacks to a large extent.

5.2 Distortive Attacks

In a distortive attack the adversary subjects the watermarked program or parts of it to some uniform distortive transformations. Since such transformations also affect the watermark cover code, the watermark may no longer be recognizable in the modified version. We are mainly interested in the semantic preserving distortive transformations, specially obfuscation. Collberg, et al. [CTL97] provide a catalog of different types of obfuscating transformations. Some of them have been derived from well-known transformations used in areas such as compiler-optimization and software re-engineering, while others have been developed for the sole purpose of obfuscation.

In the following discussion we look at some of them and how they can be used to attack SWuS and also look for counter measures.

5.2.1 Renaming Variables

In our naive scheme of recognition (refer Section 3.6), the recognizer $\mathcal{R}$ needs the name of the fragment variable to be supplied as part of the auxiliary information A_r to work properly. However, the name of all the variables in a program can easily be changed, resulting in the failure of $\mathcal{R}$.

This can be countered by taking a different approach in finding the fragment variable and corresponding watermark assignment. As part of the information A_r, the recognizer would be supplied with a set $\mathcal{V}$, where each element $(I, v, L) \in \mathcal{V}$ is a triple. I is a valid input to the subject program and v is a sequence $(v_1, v_2, \ldots, v_l)$ of the values taken by the fragment variable on the l occasions it is assigned to during execution of the (untransformed) watermarked program p_w with the input I. $L \subset [1, l]$ denotes the particular instances of v_i which are due to the execution of the watermark assignment statement.

```
for(i = 0; i < n ; i++)
{
     ...
     x = 4 * i;
     ...
     if(i % 2)
S:      x = x + 1;
     ...
}
```

Figure 9: A Code Fragment

In the program fragment shown in Figure 9, say `x` is the fragment variable and statement `S` is the watermark assignment, and there are no other assignments to `x` in the rest of the program other than the ones shown. One of the elements in $\mathcal{V}$ can be (I, v, L) where I is any input that makes n equal to 4 and $v = (0, 4, 5, 8, 12, 13)$. The set $L = \{3, 6\}$, as the third and sixth elements of v are instances of assignment to `x` due to execution of the statement `S` .

During recognition, $\mathcal{R}$ runs the program with the input I and monitors the sequence of values taken by all the program variables. The variable whose values match with v for the input I for all $(I, v, L) \in \mathcal{V}$ is chosen as the fragment variable.

The watermark assignment can be located from the information in L.

5.2.2 Introduction of Aliases

The above scheme can be defeated by the introduction of *aliases*. If two or more expressions denote the same memory address, we say that the expressions are aliases of one another ([AU86]). For example, say `x` is the fragment variable, and the original code after watermarking be as shown in figure 10. The code fragment can be changed by an adversary by renaming the variable to `y` and introducing `p`, a pointer to `y`, such that `*p` is an alias of `y`, as shown in Figure 11.

```
int x;
...
x = E1;
...
x = E2;
...
x = E3;
```

Figure 10: Code after watermarking.

```
int y;
int *p = &y;
...
y = E1;
...
*p = E2;
...
*p = E3;
```

Figure 11: Code after variable renaming and introduction of alias.

One solution is to do static *alias analysis* ([AU86]) and while monitoring the values of `y`, also take into consideration the *possible* aliases of `y`.

However, we can do better, by using a slightly modified version of the scheme introduced in the earlier section. Instead of the set $\mathcal{V}$, we can provide the set $\mathcal{V}'$ to the recognizer $\mathcal{R}$ as part of A_r. $\mathcal{V}'$ differs from $\mathcal{V}$, in that, for each triple $(I, v', L) \in \mathcal{V}'$, v' is the sequence of the values assigned to a set of *addresses* A for the input I. A is the set of different addresses of the fragment variable `x` during execution of p_w with input I. If `x` is a global A is singleton. If `x` is the local variable of a function, say `bar`, A contains all the addresses (on the stack) assigned to `x` across the different calls to `bar`.

The recognizer now maintains the history of the address(es) being written to and the value assigned while executing the program under inspection. Such history for addresses of global variables and also those allocated from the heap are to be maintained throughout the program execution (or till it is freed in case of a heap allocated location). The addresses on the stack are interpreted as the tuple (*fname*, *offset*) where *fname* is the name of the function and *offset* is the stack-frame offset of the address. All stack addresses which map to the same (*fname*, *offset*) tuple are considered equivalent and the values assigned to them are merged in the same sequence.

Note that these can be done either using a debugger or instrumenting the code properly. Finding the fragment variable now requires a matching of the observed sequence of values for these addresses for an input I with the corresponding sequence v'.

It should be kept in mind that the program source should be compiled with the register allocation option switched off, both during embedding and recognition, so that all writes to variables are actually done to memory locations.

5.2.3 Adding Spurious Definitions

An obvious attack to the scheme developed in the Sections 5.2.1 and 5.2.2 is to add some spurious definitions of the *suspected* fragment variable without affecting program behavior as shown in Figures 12 and 13. This would change the sequence of values observed by the recognizer and not match with the sequences passed in $\mathcal{V}'$.

Such code can be detected and removed by $\mathcal{R}$ through *dead code elimination*. However, if false dependencies are established between the spurious code introduced and the existing code, any form of static analysis to remove the added definitions will fail.

To counter this, the recognizer can look at the *Levenshtein or edit distance* between the sequence v' and that of the observed sequence for a given address (or address set). The edit distance measures the cost of converting one sequence into another of possibly different length by assigning costs for replacing an element as well as insertion and deletion.

The recognizer calculates the edit distance (with cost of replacement and deletion being 1 and that of insertion being 0) between the sequence v' and the observed sequences. For example, the sequences $S_1 = \{10, 4, 5, 20\}$ and $S_2 = \{10, 4, 14, 5, 19\}$ have an edit distance of 1, as we can edit S_1 to S_2 by inserting the element 14 in S_1 and replacing 20 by 19.

The address (or address set) for which the observed sequences of values assigned give the *minimum average* edit distance from the sequences v', the average being taken over the edit distances obtained for the different inputs I, for all $(I, v', L) \in \mathcal{V}'$, is chosen as the fragment variable address.

```
for(i = 0; i < n ; i++)
{
    ...
    x = 4 * i;
    ...
    if(i % 2)
S:      x = x + 1;
    ...
}
```

Figure 12: Code after watermarking

```
for(i = 0; i < n ; i++)
{
    ...
    x = 4 * i;

    z = x;
    x = z * z;
    x = z;

    ...
    if(i % 2)
S:      x = x + 1;
    ...
}
```

Figure 13: Code after adding spurious definitions

5.2.4 Sequential Code Reordering

Reordering the sequence of program statements can be done as a distortive transformation. Since we're only interested about semantic-preserving transformations, we can assume such reordering would be done taking the data and control dependencies in consideration, and hence in no way change the effect of the watermark cover code.

However, this will invalidate the information about the location of the watermark predicate and the watermark assignment provided to our recognizer $\mathcal{R}$ as part of A_r. Under such circumstances, locating the watermarking assignment would not be a problem if the scheme elaborated in the previous sections is implemented. But the watermark predicate now has to be identified by its guard tag, the opaque predicate. Since $\mathcal{R}$ knows the opaque predicate used as the guard tag, it can scan through the program to find it.

If the sequential code reordering is used along with variable renaming, $\mathcal{R}$ can still find the guard tag through pattern matching. For example, if the added opaque predicate is 7*x*x - 1 != y*y, and it is changed to 7*abc*abc - 1 != efg*efg, the recognizer can look for the pattern 7*<id>*<id> - 1 != <id>*<id> in the program to find the guard tag and hence locate the watermark predicate.

5.2.5 Function Inlining

Inlining, an important compiler optimization, can also be used for obfuscation [CTL97]. A function call in a program can be inlined by replacing it with the body of the called function. Let us examine how such inlining of functions can affect our watermark or its recognition.

The semantics of the watermark cover code can not be disturbed by inlining. However, recognition of the watermark becomes a bit more involved. Inlining can happen inside the function containing the watermark predicate and the watermark code, or this function itself can be inlined.

In the former case, identifying the fragment variable and the watermarking assignment will be unaffected if we follow the value tracking scheme developed earlier.

In the latter case, the obvious change in scenario is that now we can have more than one instance of the watermark cover code, including the watermark predicate and the watermark assignment. To counter such attacks, the recognizer $\mathcal{R}$ should be equipped to find out all of them. Clearly, when there are more than one copy of the watermarking assignment, no single statement would match the set of values in v' as listed in L in the triple $(I, v', L) \in \mathcal{V}'$, for the input I. Instead this would be matched by a set of (identical) statements in the program. The set of watermarking statements would be the union of all these sets obtained for input I, for all $(I, v', L) \in \mathcal{V}'$. Finding all the instances of watermark predicate boils down to finding all the instances of the guard tag, which can be done without any major modification of $\mathcal{R}$.

Finally, the recognizer has to try all the slices corresponding to the multiple occurrences of the watermark assignment and execute all of them with the set of watermarking inputs, until the conditions for watermark construction are satisfied, by any one of them.

5.2.6 Function Outlining

Outlining [CTL97] means turning a sequence of statements into a separate function, and then replacing those statements with the function call with the necessary arguments. It can be used to obfuscate a watermarked program, with a view to remove the mark.

Outlining can proceduralize whole or parts of the watermark cover code. In the former case the watermark recognition is unaffected, only it happens in a different function than the original watermarked program. For the latter, it might so happen that the cover code is split across procedures, e.g., the watermark predicate is in one function and the watermark assignment is in a different function which is called in the block guarded by the watermark predicate. However, such transformation cannot prevent $\mathcal{R}$ from identifying both of them, albeit from different functions.

5.2.7 Loop Transformations

Loop transformations like *loop unrolling*, *loop fission* and *loop blocking* may be applied to the program either as optimization measure or for the purpose of obfuscation.

Loop unrolling replicates the body of a loop one or more times. If the cover code resides in the body of loop, unrolling of the loop may cause multiple instances of the watermark predicate and watermark assignment, creating a scenario similar to the one discussed for function inlining (Section 5.2.5), and can be dealt with in the same way.

Loop fission, which breaks down the body of a loop into a number of loops in the same iteration space, and loop blocking, which breaks down the iteration space of a loop for improved cache behavior may also pose very similar problems and hence can be handled by $\mathcal{R}$.

5.2.8 Repackaging of Variables

Another attack that can be launched by an adversary is repackaging the simple variables of p_w into an array or as structure members, hoping to make the identification of the fragment variable impossible. However, with the value tracking scheme of $\mathcal{R}$ in place, such repackaging poses no more problems than renaming variables or introducing aliases does.

5.2.9 Conversion of Static to Procedural Data

Static data, like a character string or an integer constant may be converted into a procedure which would produce the data as its output. Such proceduralization may destroy static data watermarks. However, in our case it does not matter whether the fragment variable gets the values assigned from some static data or from the return value of a procedure as long as the assigned value is unchanged (which should be the case for a semantics preserving transformation).

5.2.10 Variable Splitting and Merging

A variable Z of type U can be split into k variables $z_1, z_2, \ldots, z_k$ as $Z = (z_1, z_2, \ldots, z_k)$. Such splitting can be done provided we can have:

- A function f that maps the values of $z_1, z_2, \ldots, z_k$ into the corresponding value of Z.
- A function g that maps the value of Z into the the corresponding values of $z_1, z_2, \ldots, z_k$.
- New operators corresponding to the primitive operators on variables of type U.

Variable merging is the reverse, where more than one variables are accommodated in a variable of a larger sized type.

Both variable splitting and merging can confuse our recognizer $\mathcal{R}$ as it would no longer be possible to find the fragment variable. However, since we are using variables which are already in use in the subject program, in the absence of the knowledge of which variables are used as the fragment variables, an attacker has to use the transformations extensively on a large number of variables in the program. Since variable splitting and merging requires complex functions to replace even the basic operations on them, such extensive use of these transformations would take a major toll on the runtime efficiency of the program which would be a major deterrent to the attacker to use them.

The complexity of the conversion and operator functions would be even more when the variable types used for splitting and merging are real. So to discourage this attack further we can use real type variables as fragment variables.

Finally, if variable splitting and merging are used only sparingly, this would mean only a small fraction of our watermark fragments would be unrecoverable. Due to the redundancy added to the watermark as proposed in Section 4, the recognizer would have a very high probability of still being able to recover the watermark.

5.2.11 Predicate Toggling

Predicates can be toggled in a program as shown in the Figures 14 and 15, without changing the semantics.

```
if(P)
    { /* Action_A */ }
else
    { /* Action_B */ }
```

Figure 14: Original Code using predicate P

```
if (!P)
     { /* Action_B */ }
else
     { /* Action_A */ }
```

Figure 15: Code after toggling predicate P

If the watermark predicate `WP` is so toggled, the recognizer will still identify it, but will be tricked into believing `(!WP)` to the watermark predicate. This will jeopardize the watermark construction as in the subsequent recognition phase it will change `(!WP)` to `(!WP || 1)` and control will never get to `Action_A` which contains the rest of the watermark cover code.

We add the set $\mathcal{B}$ to the auxiliary information A_r provided to the recognizer $\mathcal{R}$. An element $(I, b) \in \mathcal{B}$ is a tuple where I is a valid input and b is the bit string $(b_1, \ldots, b_m)$ where $b_i = 1$ if the watermarking predicate `WP` is evaluated to true in its i^{th} evaluation when p_w is run with the input I, and $b_i = 0$, otherwise.

During recognition once the watermarking predicate has been identified (say `WP'`) by $\mathcal{R}$, it executes the program under inspection with the input I and matches the first m instances of the truth value of `WP'` with the bit string b. If it is a match, then `WP'` is same as the original watermark predicate `WP`. Otherwise, if the observed bit string is the complement of b, then `WP'` is the complement of `WP`. In the latter case, instead of changing `WP'` to `(WP' || 1)`, $\mathcal{R}$ changes it to `(WP' && 0)`.

5.3 Additive Attacks

In an additive attack the adversary adds his own watermark to the program such that the temporal precedence of the original watermark is no longer detectable. Additive attacks can only be prevented by having trustworthy authorities to register the embedding time of the watermark.

6 Properties of the Watermarking method

A number of properties has been enumerated in [CT99], [NCT02] and [CC04] that should be possessed by a software watermarking method. We discuss some of these properties and how SWuS fares in these aspects.

6.1 Program Functionality

A prerequisite for a watermarking method $\mathcal{W} = (\mathcal{E}, \mathcal{R})$ to be used is that it should preserve the functionality of the subject program p. So, if

$$\mathcal{E}\,(p, w, A_e) = p_w$$

the semantics and hence the *observable behavior* of p should be retained by p_w. To be more precise, as [CTL97] puts it, the following conditions should hold:

- If p fails to terminate or terminates with an error condition for a given input, then p_w may or may not terminate for that input.
- Otherwise, p_w must terminate and produce the same output as p.

In SWuS, the slice assignment statement `S` is replaced by the watermark code in such a way that whenever `S` was to be executed in p, the action pair

$(A_k, B_k), k \in [1, n]$, is executed in p_w. Given the properties of these action pairs, the effect on the fragment variable x and hence the program output is same. However, while designing the watermark code, care should be taken that there are no undesired side effect due to overflow, division by zero, etc.

6.2 Program Performance

The watermarking method should not hamper the performance of the program in terms of time and memory consumption. Our method of watermarking adds very little memory overhead as it mostly uses variables already present in the subject program p. The execution time is increased, though, by a small amount as every execution of `S` now maps to two function calls and the extra code executed in them. This effect can be kept at minimum by:

1. The slice assignment `S` in p is chosen such that it is not present in a program *hotspot*, that is, a frequently executed part of the program, such as loops.

2. The actions $\{A_1, \ldots, A_n\}$ and $\{B_1, \ldots, B_n\}$, and the predicates $\{P_1, \ldots, P_n\}$ and $\{Q_1, \ldots, Q_{n-1}\}$ should not be too complicated.

3. Additionally, the predicates in P should be so arranged that the probability of P_{i_1} being true is more than that of P_{i_2} where $i_1 < i_2$. Since, $P_k \Leftrightarrow Q_k$, same would hold for the predicates in Q. This would imply that *in average* lesser number of predicates are actually evaluated in the watermark functions before finding the true one during program execution.

6.3 Credibility

Not only should the recognizer be able to extract the watermark correctly from a watermarked program, it should also not produce any false recognition as mentioned in Section 2. This means, the recognizer $\mathcal{R}$ should not extract a watermark from a program that has, in the first place, never been watermarked by the embedder $\mathcal{E}$. Otherwise, the credibility of $\mathcal{M} = (\mathcal{E}, \mathcal{R})$ would be at stake.

Assuming that the watermark key (A_e, A_r) provided to $\mathcal{E}$ and $\mathcal{R}$ is kept secret, it is extremely unlikely that an independent program p' (that is, which is not a transformation of a watermarked program p_w) can exist, either by malicious intent or by accident which would result in a false recognition.

6.4 Watermark Stealth

The stealth of a watermarking method is a measure of how imperceptible it is to an observer who might subject the watermarked program to a variety of analysis, in the absence of the recognizer $\mathcal{R}$. If the watermark cover is identified, there is a very high probability of launching a subtractive attack and removing it. For example, if the dynamic graph encoding method discussed in [CT99] is used in numerical programs, the embedded code for the construction of graph is readily visible.

SWuS gives the flexibility of molding the cover code to *mix* well with the original code, the only constraint being that it should satisfy the properties elaborated in Section 3.5. Also, the use of variables already in use in the original program to hold the watermark fragments adds to the stealthiness of the method.

6.5 Watermark Resilience

The resilience of a watermarking algorithm is its degree of immunity from different attacks. It should be impossible or at least extremely hard or require a huge amount of time to be invested by an attacker to locate, remove or render useless the watermark structure from a watermarked program without hampering the usefulness of the program to a great extent.

SWuS is resilient to most of the automated semantic preserving attacks as has been discussed in details in Section 5.

6.6 Multiple Watermarks

A watermarking method should not be susceptible to re-watermarking. In the same vein, it should be possible to embed multiple watermarks in a subject program, and still be able to recognize each of them. Formally, embedding a new watermark should not make the old watermark unrecognizable, that is,

$$\begin{gathered}
\text{if } p_w = \mathcal{E}\left(p, w_1, A_{e_1}\right) \\
\text{and } \mathcal{R}\left(p_w, A_r\right) = w_1 \\
\text{then } \mathcal{R}\left(\mathcal{E}\left(p_w, w_2, A_{e_2}\right), A_r\right) = w_1 \\
\text{where } w_1, w_2 \in \mathcal{W} \text{ and } A_{e_1}, A_{e_2} \in \mathcal{A}_{\mathcal{E}}
\end{gathered}$$

Slice-based watermarking can be used for multiple watermarks. The only restrictions are that the guard tags and slice assignments should not be reused.

6.7 Fingerprinting

Fingerprinting differs from watermarking in that it requires inserting a unique mark to be embedded in each copy of the subject program. This way not only is software theft detected, but also the source of such infringement can be traced.

SWuS lends itself without any modification for fingerprinting. However, it is to be noted that fingerprinting can make a watermarking scheme prone to collusive attacks, where different copies of the fingerprinted program are compared to locate the watermark. If the watermark cover code is obfuscated differently in these fingerprinted versions such attacks can be resisted.

6.8 Unbounded Watermark Size

The watermarking algorithm should, ideally, not bound the size of the watermark value embedded. In SWuS the embedded watermark can be arbitrarily large. This is achieved by the use of the Reed-Solomon encoding and the subsequent splitting up of the watermark in parts. Increasing the watermark size only results in a larger number of watermark fragments to be embedded. For example, in the RS code proposed in Section 4, increasing the watermark size from 8 to 16 bytes, would mean 48 fragments to be embedded instead of the earlier 40 watermark fragments.

7 Functional Requirements of a Watermarking Tool

In order to develop a tool based on a watermarking algorithm $\mathcal{M} = (\mathcal{E}, \mathcal{R})$, $\mathcal{M}$ should satisfy certain requirements as discussed in this section.

7.1 Automation

A prerequisite for developing a tool based on $\mathcal{M}$ is the recognizer $\mathcal{R}$ should require no manual intervention, or else there may be allegations of foul play. Also the embedder $\mathcal{E}$ should be as much automated as is possible. These requirements are satisfied by SWuS. The recognition process is fully automated. Though the embedding can also be fully automated, we propose that it be kept semi-automatic with some participation from the programmer(s), e.g., while choosing the watermarking assignment avoiding program hotspots or some extremely sensitive or optimized part of the program. Programmer participation is also encouraged during manufacturing the name of the new functions introduced, the predicates and actions, as also to monitor the stealth of the cover code generated and check for its possible vulnerabilities. Only a well balanced tool-programmer collaboration would ensure a watermark that satisfies the properties discussed in the Section 6.

7.2 Making the Watermarking Algorithm Public

Both the watermark embedder and the recognizer algorithm in SWuS can be made public. This is possible because they need the watermark key (A_e, A_r) to perform their task, and this is kept secret. So just having the possession of the recognizer won't help an adversary in finding the watermark, till he gets access or guesses (which is extremely unlikely) the corresponding watermark key. Making the recognizer public is all the more necessary in our scheme as it makes some changes to the extracted watermark slice, and people should be convinced that these changes are trivial and generic, and in no way specific to the program under inspection.

7.3 Cost

Embedding the watermark into the subject program is an one time investment and hence the cost in both developer time as well as the time taken by the tool to actually make the required modifications is not very critical. However, this should not be exceptionally high. In SWuS this cost is not much in terms of programmers' time. Also the time taken by the embedder is comparable to either the compilation time or the execution time of the program depending upon how the tool is actually implemented.

The time taken by the recognizer is bounded by the maximum of compilation and execution time. [NCT02] mentions that sometimes it is desirable to have the recognizer to work slowly to prevent *oracle attacks*. An oracle attack is where an adversary has access to the recognizer and makes small changes to the software until the watermark recognizer fails. However, since our recognizer needs the auxiliary information A_r, which is kept secret, launching such an attack is not very feasible.

8 Implementation

We have developed a tool for automated slice-based software watermarking on the `gcc` framework. The tool implements all the fundamental features of SWuS along with the encoding scheme proposed in Section 4 and the value tracking mechanism to counter variable renaming as mentioned in Section

5.2.1. It is to be noted though SWuS has been developed from the concept of program slices, any explicit slicing is not at all required in an implementation.

The watermark embedder in the tool expects the subject program and the watermark as input as also the specification of the auxiliary information A_e in the form of the fragment variables, slice assignments and guard predicates.

The watermark recognizer component of the tool takes a possibly transformed version of the watermarked program and with the help of the auxiliary information A_r, which includes the specification of the guard tags, tries to extract the watermark.

9 Summary And Future Work

In this section, we first summarize the main contributions of this work. Next, we offer several suggestions for the future research.

9.1 Summary

Software watermarking is slowly being recognized as a promising mechanism to curb illegal copying of software. However, software watermarking cannot stop software piracy. The main aim of watermarking is to enable the author of the software to prove ownership rights in case intellectual property rights violation takes place.

We have proposed a slice-based watermarking method, SWuS, which hides the watermark in an unreachable state of a program slice. The watermark code added to the subject program is extremely customizable to gel well with the nature of the original program and also has very low runtime overhead. It has been shown that the scheme, with adequate enhancements, is also resilient to most of the automated attacks possible on a watermarking scheme, even though both the embedding and recognition algorithm is made public, given that the watermarking key is not revealed. Finally, the Reed-Solomon code has been used in the scheme to add redundancy to the embedded watermark that adds to its robustness, makes possible to recover the watermark from only parts of the program, and also provides the provision for an arbitrarily large sized watermark.

9.2 Future Work

Future work can concentrate on finding more resilient methods to identify the watermark predicate from a possibly transformed watermarked program than the opaque predicate guard tag proposed in this report. Also the recognition phase of SWuS presently works on the source code. Research can be done to suitably modify the scheme to also work on program binary.

References

[Arb02] Genevieve Arboit. A method for watermarking java programs via opaque predicates. In *Proceedings of Fifth International conference on Electronic Commerce Research*, 2002.

[AU86] Ravi Aho, Alfred V.and Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[BGI+01] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S.P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Electronic Colloquium on Computational Complexity (ECCC)*, 2001.

[BSA03] Eighth annual bsa global software piracy study: Trends in software piracy 1994-2002. Business Software Alliance, June 2003.

[CC04] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Principles of Programming Languages, POPL 2004*, Venice Italy, Jan 2004.

[CCD+04] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *SIGPLAN '04, Conference on Programming Language Design and Implementation*, June 2004.

[CHC03] D. Curran, N.J. Hurley, and M.O. Cinneide. Securing java through software watermarking. In *Proceedings of the 2nd International conference on Principles and practice of programming in Java, PPPJ 2003*, Ireland, June 2003.

[CT99] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Symposium on Principles of Programming Languages*, San Antonio, Jan 1999.

[CTL88] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, January 1988.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. In *Technical Report 148, Dept. of Computer Science, University of Auckland*, July 1997.

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

[EEG] The easter egg archive. http://www.eeggs.com.

[JR00] M. Jakobsson and M. Reiter. Discouraging software piracy using software aging. In *Proc. ACM Workshop Security and Privacy in Digital Rights Management*, 2000.

[MC96] Scott A. Moskowitz and Marc Cooperman. Method for stega-cipher protection of computer code. US Patent 5,745,569, January 1996. Assignee: The Dice Company.

[MIM+04] Akito Monden, Hajimu Iida, Kenichi Matsumoto, Katsuro Inoue, and Koji Torii. A practical method for watermarking java programs. In *24th Computer Software and Applications Conference, Compsac '2000*, Taipei, Taiwan, 2004.

[NCT02] Jasvir Nagra, Christian Collberg, and Clark Thomborson. A functional taxanomy for software watermarking. In *25th Australasian Computer Science Conference, ACSC '2002*, Melbourne, Australia, January 2002.

[RS60] I.S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2), June 1960.

[SHKQ99] J.P. Stern, G. Hachez, F. Koeune, and J.J. Quisquater. Robust object watermarking: Application to code. In *Workshop on Information Hiding*, 1999.

[TNSH] Clark Thomborson, Jasvir Nagra, Ram Somaraju, and Charles He. Tamper-proofing software watermarks. In *Australasian Information Security Workshop, AISW 2004*, Dunedin, New Zealand.

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

[Wei84] M. Weiser. Program slicing. In *IEEE Transactions on Software Engineering*, July 1984.