The Department of Computer Science
The University of Auckland
New Zealand

Unauthorized Detection of CT Watermarks Based on Pattern Analysis Methods

Teng Teng

July 2006

Supervisors:

Clark Thomborson

A thesis submitted in partial fulfillment of the requirements of Master of Science in Computer Science

The University of Auckland

Thesis Consent Form

This thesis may be consulted for the purpose of research or private study provided that due acknowledgement is made where appropriate and that the author's permission is obtained before any material from the thesis is published.

I agree that the University of Auckland Library may make a copy of this thesis for supply to the collection of another prescribed library on request from that Library; and

1. I agree that this thesis may be photocopied for supply to any person in accordance with the provisions of Section 56 of the Copyright Act 1994.

   Or

2. ~~This thesis may not be photocopied other than to supply a copy for the collection of another prescribed library.~~

   (*Strike out 1 or 2*)

Signed: .....

Date: .....

Created: 5 July 2006

Last updated: 14 July 2006

Abstract

The CT watermark algorithm is a relatively new software watermarking algorithm. To evaluate the robustness of the CT watermark algorithm, we tried to simulate the unauthorized detection of watermark attacks on the CT watermark. The main concern of this thesis is to explore the effect of the watermark size ratio of watermarked programs on the accuracy of attackers' detection function.

We simulate attackers by using the statistical pattern classification method to detect the existence of a CT watermark. The adversaries' behaviour is confined by a set of rules proposed by the CT watermark algorithm designers, which I extend. We judge the success of attackers by the false positive and false negative rates of attackers' detection function.

Results showed that when the watermarked programs that the attackers used to train their detection function and the watermarked programs to be detected both have a high watermark size ratio, the attackers' detection function will be accurate. So restricting the watermark size ratio of CT watermarked programs should increase their robustness.

We suggest two ways to amend this problem. One way is to use the numeric watermark option in the watermark embedding procedure to reduce the watermark size ratio. Another way is to denote those programs which will cause a high watermark size ratio as being unsuitable to be CT watermarked.

Acknowledgement

I would like to thank Prof. Clark Thomborson, my supervisor, for his encouraging and constructive suggestions. This thesis is a continuous study based on his and Christian Collberg's unpublished work which was submitted to TOPLAS in 15 April 2006.

Thanks for Barbara Thomborson and Dr. Stephen Drape who taught me quite a lot about thesis writing. Stephen also guided me on Latex and mathematical notation. Thanks to Jasvir Nagra offering me advice for annotating Java programs. Thanks for Dr.Collberg for providing me his collection of Java programs. I also appreciate other group members' useful suggestions.

Contents

1

Introduction

1.1 The CT Watermark

A watermark is an indicator added into an intellectual product to prove the ownership of that product [20, 26, 10, 12, 11]. The watermark embedded in software is called a *software watermark*. The main categories of software watermark algorithms are *static* and *dynamic*. The static watermark algorithm directly embeds watermarks into software. The embedding or extracting procedures of static watermark do not require the execution of a program. In contrast, dynamic watermark algorithms only embed codes which construct watermarks at program run time [10]. The benefit of dynamic watermark algorithms over static is that "dynamic watermarking techniques are more stealthy and more resilient than the existing alternative technology of static watermarks" [10].

As a dynamic watermark algorithm, the CT watermark algorithm embeds "a watermark in the topology of a dynamically built graph structure" [9]. The main benefit of the CT watermarking algorithm over other dynamic watermark algorithms is that it uses graph structures to represent watermark. The reason for using graph structure is that point aliasing effects make it difficult to locate codes that build the graph structure. As a result, statistically analyze CT watermark will be troublesome for adversaries [11]. In addition, the designers claim that CT watermark algorithm will not be affected by some of those transformations occurring from translations, optimizations and obfuscations [9]. These transformations are harmful for static software watermark algorithms and some dynamic watermark algorithms other than CT algorithm.

Other defense mechanisms of the CT watermark algorithm are also employed. For example, the CT watermark algorithm use cycle graph to resist node splitting attacks [11]. "Node splitting attack" means that adversaries try to split nodes into several linked components to distort the watermarking graph. [11].

Three desiderata are often used to evaluate the watermark[11]:

- ***Stealth***: Stealth is the ability of a watermark algorithm to hide watermark. It is the measurement on how difficult for adversaries to detect a watermark.

- ***Resilience***: Resilience is the ability of a watermark to resist possible attacks. It is the measurement on how easy for adversaries to attack a watermark successfully.

- ***Data Rate***: *Data rate* is the ratio of the entropy of w to "extra" program size caused by watermark embedding[8]. It is the measurement on the efficient of watermark. In this thesis, we concern more on *watermark size ratio* which is closely related to data rate. *Watermark size ratio* is the ratio of 'extra" program size to the cover program size. It is the measurement of the capacity of watermark embedding. If the watermark, the cover program and watermark embedding parameters are all the same, then *data rate* is inverse related to *Watermark size ratio.* and A formal definition of watermark size ratio is given in Section 4.6.

Sometimes, to improve one desideratum will worsen other desiderata. So watermark design must consider the trade-off between those desiderata. [11]

One of the most important considerations in the design of the CT watermark algorithm is its resilience. We will discuss possible attacks to verify the *resilience* of the CT watermark algorithm in next section.

1.2 Possible Attacks

Collberg [11] suggests some types of possible attacks on CT watermark. Attack types suggested for other watermark algorithms [12] may also be applied on CT watermark algorithm. We summarize categories of possible attacks on CT watermark as follows.

- Unauthorized detection of watermark attack (will be abbreviated as AUDW afterwards)
- Additive attack
- Distortive attack
- Subtractive attack
- Protocol attack

AUDW tries to detect secret information about watermark without permission. Cox [12] suggests three types of unauthorized watermark detection threats:

Type I: Attackers can decode the watermarks embedded

Type II: Attackers can detect the existence of watermarks without decoding the watermark

Type III: Attackers can detect the existence of watermarks, but they can not decode the watermark. However, they can distinguish whether two watermarks share the same key.

Additive attack: Attackers try to embed a bogus watermark by the attacker into software and use it as the basis of their own claim on the software [11].

Distortive attack: Attackers try to transform the watermark with or without transforming the program as well. The goal of distortive attacks is to distort a watermark so that the authorized watermark recognizer can not longer recognize it [11].

Subtractive attack tries to remove the watermark embedded in software without degrading the quality of software [11].

Protocol attack tries to build a bogus watermark recognizer which can recognize the bogus watermark embedded by the attack [13]. Then they can claim their fake ownership to the potential buyer.

In this thesis, we will focus on type II of AUDW. We choose to focus our attacks against the CT watermarking algorithm because it is designed to resist such automated attacks [11]. In addition, type II attacks can be the basis of other attacks such as subtractive attack and distortive attack.

1.3 Motivation of This Research

Since the CT watermark is a relatively new approach compared to other watermark algorithms, little in-depth research exists on attacks to CT watermark. In this research, we attempt to determine the robustness of the CT watermark to AUDW. We want to determine attackers' accuracy in detecting the CT watermark. Specifically, we chose to explore the effects of the watermark size ratio on accuracy. We also consider the impacts of feature selection on attackers' accuracy in detecting the CT watermark.

One goal in our experiment is to explore how the watermark size ratio affects the accuracy of attack. Since the data rate is in indirect proportion to the watermark size ratio,

Among the three desiderata used to evaluate the CT watermark, *resilience* is an essential metric for assessing the success of watermark algorithms [11]. Since the attack we implemented is a detection of watermark attack, the *stealth* of the CT watermark must also be evaluated. Lastly, we consider how the data rate relates to the error rate of AUDW.

In order to conduct our experiment, we need to implement some attacks on CT watermark algorithm. In addition to our direct experimental measurement, we can gain understanding of how well those defense mechanisms of the CT watermark algorithm work. What is more, we can also check whether those defense mechanisms will introduce other vulnerabilities. For example, as we mentioned in Section 1.1, one defense mechanism of the CT watermark algorithm is using cycle graph to avoid node splitting attack. However, cycle graph using much more nodes than uncycled graph. Obviously, it will reduce the stealth of the CT watermark. Finally, our ultimate goal is to improve the CT watermark algorithm by analyzing those methodologies that can be employed to attack the CT watermark successfully.

In AUDW, attackers try to develop their own $\mathfrak{detect}$ function based on attack methods and/or the watermarked and unwatermarked programs they collected. Then they can report whether a program is watermarked or not by their $\mathfrak{detect}$ function. A more detailed AUDW attacker model are given in Chapter2.

The purpose of this thesis is to implement some AUDWs on the CT watermark based on statistical analysis methods. These AUDWs will be used to evaluate the stealth of the CT watermark. The detection of the CT watermark can be unauthorized because the CT watermark algorithm is a steganographic watermark algorithm. A steganographic watermark algorithm is designed not only to hide the content of watermark but also the existence of watermark [11]. The reason that we start from AUDW is given in last subsection. We limit the methods we used in our detection to statistical analysis methods because statistical analysis methods can be implemented automatically. Since CT watermark designers try to design their watermark algorithm against attacks can be automatically implemented.

The method we use to detect the CT watermark is pattern classification. Pattern classification is to distinguish the CT watermarked programs from unwatermarked programs based on the "pattern" (the measurements of a set of characteristic features of an object [40]) of programs. This method can find out whether the frequencies of features of a CT watermarked program are "abnormal" enough to distinguish it from most other

unwatermarked programs.

1.4 Related Work

Currently, although few methods to attack the CT watermark are practically implemented, several attacking methods in related fields may also be applied on the CT watermark. Those related fields include those attacks on media watermark [22] [37] or steganographic systems [31]. Other possible useful detection methods include those methods used in virus detection [34] and software birthmark detection [25]. In [11] [30], several possible attacks on the CT watermark are also suggested. In addition, we also learn from the information hiding and model of attackers setting issues discussed for media watermark [38, 5, 24].

Collberg and Thomborson [11] discuss the evaluation of stealthy of the CT algorithm by using feature occurrences. Their work is very related to this thesis. However, their experiment just gives some evaluation results from the viewpoint of defenders. This thesis tries to implement some unauthorized detection of watermark attacks on the CT watermark so it is evaluation from the viewpoint of attacker. Maybe our research can be viewed as a consecutive research of [11].

1.5 Structure of This Thesis

In this Chapter, we have briefly overviewed the CT watermark algorithm and possible attacks on CT watermark. We realize that the unauthorized detection of CT watermark is a kind of attack because CT watermark algorithm is a steganographic watermark algorithm.

In Chapter 2, we set up the attacker model for AUDW. Based on the designer's limitations on attackers, We defined the rules to limit the behaviours and to judge the success of AUDW attacker. We assert that the success of attacker is decided by the accuracy of attacker's $\mathfrak{detect}$ function. Additionally, we analyze the designers' limitations on attacker and argue that attackers should be given less information of CT algorithm.

In Chapter 3, we present the pattern classification theory. We assume that attackers understand the pattern classification theory and will try to use the theory to optimize his $\mathfrak{detect}$ function. We survey factors can theoretically affect the accuracy of attacker's $\mathfrak{detect}$ function in this chapter.

In Chapter 4, we explain the design of our experiment. We argue that our experiment design basically conform the pattern classification theory with some minor violations.

In Chapter 5 we present our experiment results and analyze the meaning of them.

In Chapter 6, we make the conclusion of our thesis and give some suggestions on our future work.

In Appendix A.1, we offer the information on our experiment environment which includes the hardware and softwares we used in this experiment.

In Appendix A.2, we present the Java opcode set, the sample set and the watermark set used in our experiment. The default parameters used in our watermark embedding procedure are also listed in Appendix A.2. The syntaxes of the discriminant function of SPSS used in our experiment are recorded and preserved in Appendix A.2.

2

The Game Between the Attacker and the Defender of the CT Watermark

2.1 Introduction

The general focus of this chapter is on describing the CT algorithm model and the AUDW model for later discussion. We also discuss the restrictions on the CT watermark, e.g.designers' limitations on attackers. In our experiments, we assume that the attackers of the CT watermark will not violate those restrictions. Lastly, the criteria to judge the success of attackers and/or defenders will be given and discussed.

2.2 The CT Watermark: Defender Side

2.2.1 CT Watermark Model

Currently, the CT watermark is implemented for Java programs [11]. Our experiments are all based on Java programs.

Definition (Program Set)

Let $\mathcal{P}$ be the set of all the Java programs. $\mathcal{P}_u$ is the subset of $\mathcal{P}$ which consists of all the Java programs which are not watermarked by the CT algorithm. $\mathcal{P}_w$ is the subset of $\mathcal{P}$ which consists of all the Java programs which are watermarked by the CT algorithm.

$$\mathcal{P} = \mathcal{P}_w \cup \mathcal{P}_u$$

We denote a program as p , $p \in \mathcal{P}$, an unwatermarked program as p_u, $p_u \in \mathcal{P}_u$ and a watermarked program as p_w, where $p_w \in \mathcal{P}_w$.

Definition (Watermark Set)

Let $\mathcal{W}$ be the set of all the watermarks which can be used in the CT watermarking algorithm, where $\mathcal{W} \subseteq \mathbb{P}\,\mathsf{String}$ [15], and $\mathbb{P}\,\mathsf{String}$ is all possible *string*s. A *string* is a sequence of characters such as "a", "1","A". We denote a watermark as w, where $w \in \mathcal{W}$.

The designers of the CT watermark algorithm define their CT watermark model with following functions [11]:

$$\mathfrak{embed}_{CT}(p, w, key) \rightarrow p_w$$

$$\mathfrak{extract}_{CT}(p_w, key) \rightarrow w$$

$$\mathfrak{recognize}_{CT}(p_w, key, w) \rightarrow [0.0, 1.0]$$

The $\mathfrak{embed}$ function embeds the watermark w into a program p by using a secret key. The key is a sequence of inputs from the user. The $\mathfrak{extract}$ takes out w from p_w to prove the ownership of the program p. The program p and its watermarked version p_w should

exhibit the same input-output behaviour. The embedding of a watermark should have little or no adverse effects on the performance of the program p. That is, p_w should execute at almost the same speed as p. p_w also should not have significantly more bugs than p.

The embed function comes in three processes. Those steps include annotation, tracing and embedding [7]:

- *Annotation* indicates the locations suitable for inserting watermark-graph-generating codes in the target program. Places such as performance-critical points or conditionalized points are not ideal to put watermark-graph-generating codes in.

- *Tracing* records the list of *tracepoints* created by a *key* which is the user's secret inputs. The *tracepoints* are the selected locations for inserting watermark-graph-generating codes after running the user's secret inputs. The *key* will be used in watermark extraction. The *tracing* information determines the final locations to embed codes for constructing the watermark.

- *Embedding* means the process of inserting watermark-graph-generating codes in the locations determined by the *tracepoints*.

2.2.2 Security Requirement

The Security Goal of the CT Watermark

The purpose of the CT watermark is to prove the ownership of intellectual property. The security goal of the CT watermark algorithm is to increase the difficulties for attackers to implement those offensives which can be implemented automatically, such as distortive attack or AUDW. The CT watermark algorithm seeks to maximum its resilience and stealth [11].

Restrictions of the CT Watermark Algorithm

The authors of the CT watermark algorithm restrict the types of attacks that the CT watermark should resist. They assert that they do not expect to develop a perfect watermark system [11]. A perfect system should resist any non-trivial attack [11].

The CT watermark algorithm is designed to resist attacks using automated methods. Those automated methods mainly consist of code transformations such as compilation and decompilation, binary translation, optimizations, compressions and obfuscations [11].

The CT algorithm is also designed to maximize its stealth [11]. It tries to obstruct the adversaries' attempts to detect the existence of the CT watermark at all [11]. So the CT algorithm should be able of resisting AUDW.

The CT algorithm designers restrict the capabilities of adversaries who will attack the CT watermark as well. Their limitations on the capabilities of adversaries are as follows [11]:

- designers' limitation 1 on attacker: Attackers can obtain all information of the CT watermark algorithm. But they can not obtain the content of watermark and/or the key to build and extract the CT watermark directly from the CT watermark owner or defender.

- designers' limitation 2 on attacker: Attackers can obtain the CT watermarked programs in the form of "a collection of Java class files" [11].

- designers' limitation 3 on attacker: Attackers will find it difficult to read the CT watermarked file manually. It means that the size of the target file should not be too small. It also means that attackers will prefer to use automatic methods.

- designers' limitation 4 on attacker: Attackers will not rewrite the watermark after studying the input-output behaviour of the watermarked program [11].

In our experiment, we introduce four additional limitations on adversaries to simplify our experiment:

- Additional limitation 1: Adversaries can only obtain one copy p_w without its unwatermarked version p_u.
- Additional limitation 2: The **embed** and **extract** function are unavailable to adversaries. So the attacker should not access Sandmark which is the tool used to embed and extract the CT watermark.
- Additional limitation 3: The attacker will only use one kind of statistical pattern classification algorithm: Fisher's discriminant function.
- Additional limitation 4: The attacker is interested only in the existence of the CT watermark (type II AUDW).

We will discuss the meaning of the four designer's limitations in Section 2.5. The four designer's limitations and the four additional limitations on the attacker will confine the behaviour of the attacker in our experiments.

2.2.3 Defensive Mechanism

The CT watermark algorithm also employs some defensive mechanisms against AUDW. Those mechanisms include [11]:

- Splitting a large graph into smaller components and spreading the codes to build those components in different locations. So the codes to build the CT watermark graph become harder for attackers to discover.
- Avoiding declaration of those root nodes constructing the CT watermark subgraphs as global variables. Good object-oriented programming will not use a lot of global variables. As a result, using global variables to declare root nodes is not stealthy. The solution is to "pass roots in the formal parameters of methods".
- Hiding the watermark class. The CT watermark algorithm must use a special class to create nodes which will construct the watermark graph. However, it is not very stealthy if the CT algorithm uses the same special class to watermark every program.

The solution in the CT algorithm is to pick an appropriate class in the program to be watermarked to build those nodes for the watermarking graph. The appropriate class here is one that fulfills the requirements for building the watermark graph. When an appropriate class is unavailable, a less desirable choice is to select an appropriate class in the Java library.

- Tamperproofing the CT watermarks by checking the predefined character of watermark graphs (i.e. checking the data in a graph data structure.). For example, the predefined character such as the diameter of a planar graph. Such checking data operations are often used in object-oriented programming. However, other watermark algorithms often use the tamperproofing method of checking the integrity of executable codes. But checking the integrity of executable codes is somewhat unusual because there is little need to do so in normal software programming. Thus the tamperproofing method used by the CT watermark is stealthier.

- Using obfuscating methods to resist a pattern-matching attack [11]. Since the CT watermark can resist many code transforming operations, many obfuscating methods will not affect the CT watermark. Instead, some obfuscating methods lend themselves as a defense in the CT watermark algorithm. Sandmark, the tool developed by Collberg for watermark algorithm research, already provides some obfuscation methods such as array splitting and class encryption. Some advanced obfuscation algorithms can be more powerful in such applications. Those advanced obfuscation algorithms include algorithms described in [6] and [39]. However, considerations of obfuscation effects on attacks are beyond the scope of this thesis.

- Using codes related to objects creating and objects linking to construct the CT watermark graph. In an object-oriented programming language such as Java, codes related to create objects and link objects are popularly used by programmers. So such codes will not be quite 'suspicious' to attackers.

2.3 CT Watermark: Attacker Side

2.3.1 Detection of Watermark Attack Models

Our detection of watermark attack model can be described by two functions: the $\mathfrak{train}$ function and the $\mathfrak{detect}$ function. The $\mathfrak{train}$ function is used to build the attacker's $\mathfrak{detect}$ function.

$$\mathfrak{train} : \mathcal{P}_{w,a} \times \mathcal{P}_{u,a} \times \mathcal{L} \rightarrow \mathcal{D}$$

Where $\mathcal{P}_{w,a}$ is a subset of $\mathcal{P}_w$, the set of watermarked programs accessible by the attacker. The attacker can know a program p is watermarked if $p \in \mathcal{P}_{w,a}$. Similarly, $\mathcal{P}_{u,a}$ is a subset of $\mathcal{P}_u$ which is the set of unwatermarked programs accessible by the attacker. The attacker can know a program is unwatermarked if $p \in \mathcal{P}_{u,a}$. $\mathcal{L}$ is the set of all pattern classification algorithms, For example, pattern classification algorithms such as Fisher's Linear discriminant algorithm or Euclidean distance discriminant algorithm. $\mathcal{D}$ is the set of all $\mathfrak{detect}$ functions based on the pattern classification method.

The attacker's $\mathfrak{detect}$ function in our thesis is an $\mathfrak{detect}$ function using the *key* to embedding the watermark.

$$\mathfrak{detect} : \mathcal{P} \rightarrow \mathbb{B}$$

Where $\mathbb{B}$ is the boolean set $\mathbb{B} = \{T, F\}$, where T means true and F means false. The function $\mathfrak{detect}$ has the following property:

$$\mathfrak{detect}(p) = T \Rightarrow p \in \mathcal{P}_w$$

$$\mathfrak{detect}(p) = F \Rightarrow p \in \mathcal{P}_u$$

An early stage of developing a formal AUDW model on the CT watermark algorithm is also explored. As Prof. Thomborson outlined in a private communication [36], "No one has published a general, information-theoretic, model of attacks on software watermarks, analogous to the attack model proposed recently for audio-visual watermarks [2]". We

still need much more effort to complete it. The immature model is as follows:

The attackers learn from the open security information $\mathcal{I}_o$ and use some software analysis skills SK_a to develop a detect function of

$$\mathcal{I}_o \times SK_a \rightarrow \mathfrak{detect}$$

Then attackers use their $\mathfrak{detect}$ to find the hidden information $\mathcal{I}_h$.

$$\mathfrak{detect}(p_x) \rightarrow \mathcal{I}_h$$

Where p_x is a program that attackers do not know if it is watermarked. $\mathcal{I}_h$ is the hidden information that the CT watermark algorithm tries to hide. In our experiment, the $\mathcal{I}_h$ that the CT watermark algorithm tries to hide is the existence of the CT watermark. We assume attackers can not judge whether p_x is p_u or p_w without using their own $\mathfrak{detect}$ function.

The $\mathcal{I}_o$ may include one or more classes of the following information, as suggested in [21] :

- The availability of the information of the CT watermark algorithm. It is the extent of details from which attackers can learn the CT watermark algorithm. In our attack model, we assume that attackers can learn all the details of the CT algorithm. The reason for accepting this assumption is discussed in the Section 2.2.2.

- The number of program p_w that the attackers can collect. In our attack model, we only consider the scenario that the attackers can collect only one copy of the p_w without the unwatermarked version of p_w. But we assume that attackers can obtain as many as other possible unwatermarked programs p_u without their p_w versions.

- The availability of $\mathfrak{embed}$ and $\mathfrak{extract}$ function. In our attack model, we consider only the the scenario that attackers can not access the official $\mathfrak{embed}$ and $\mathfrak{extract}$ function.

SK_a can be static or dynamic or hybrid static-dynamic program analysis skills. They differ mainly in that: static analysis obtains the information it is inspecting for by checking the codes of the target program without running the program; however, dynamic analysis obtains the information by executing the program. Hybrid static-dynamic analysis is a mixture of static and dynamic analysis [23]. Statistical analysis is one kind of static analysis method. When we do the statistical analysis based on the pattern of the program, then the method is called *software statistical pattern analysis*. In our attack model, we predefine the attackers own *software statistical pattern classification* skill. So they can classify softwares into classes (in our case, the classes are the CT watermarked program class C_w and unwatermarked program class C_u) based on pattern classification skill. The pattern classification skill is to classify an unknown program into C_w or C_u based on the statistically significant difference of patterns between C_w and C_u.

$\mathcal{I}_h$ is the information that the watermark designers try to hide. $\mathcal{I}_h$ may include one or more pieces of the following information suggested by Cox [12]:

- The existence of a watermark: The information about whether an unknown $p_x \in C_w$ or $\in C_u$. In our attack model, we will consider only this scenario.
- The content of watermark: The information about the secret message embedded the watermark. We will not consider this scenario in our attack model.
- Location of watermark: The information about the locations of codes to construct the watermark in the codes of target program.

2.3.2 Limited Meaning of Our Detection of Watermark Attack

The results of our attack will only have limited meaning. Firstly, we can not distinguish whether the vulnerability is from the CT watermark algorithm or its implementation To distinguish where the vulnerability comes from, we need a white-box-check, i.e. checking the codes of Sandmark. But our experiment is a black-box-test, i.e. not allowed to check the codes of Sandmark. So we can not make the judgement in our experiment. The white-box-check is an area for our future work. Secondly, we can not deny the possibility that a

smarter attacker can find a better $\mathfrak{detect}$ function than ours. So even our attack fails, we can only claim that the CT watermark is stealthy enough for our attack. No promise that the CT watermark can resist all other detection attacks. Lastly, our experiment result is only valid for our experimental sample set. So even if our attack is totally success, we still can not deny the possibility that it will fail for some programs not in our experimental sample set.

2.4 Judgement

To choose the winner between the defender and the attacker, we need a judgement rule. The judgement rule is defined by the possibility of misclassification on the attacker's part. If the attacker misclassifies more than half of the cases to be tested, then his/her $\mathfrak{detect}$ function is no better than a random guess. So the attacker fails and the defender wins completely. Otherwise the attacker at least can learn some information, so the attacker succeeds to some extent. The judgement can be more precise by introducing the false positive rate (FP) and false negative rate (FN). False positive means that the classifier falsely report an input program as watermarked when that program is actually not watermarked. False negative means the classifier falsely reports an input program that is unwatermarked when that program is actually watermarked. According to Collberg et al. [11], attackers will be more concerned with FN. The higher the FN, the higher the risk that an adversary will be "caught out". The equations for computing FP and FN are:

$$FP = \frac{|\{p_x \in C_u : \mathfrak{detect}(p_x) = T\}|}{|\{p_x : p_x \in C_u\}|} \tag{2.1}$$

$$FN = \frac{|\{p_x \in C_w : \mathfrak{detect}(p_x) = F\}|}{|\{p_x : p_x \in C_w\}|} \tag{2.2}$$

Figure 2.1 shows our judgement categories are mutually exclusive. We can describe the failure of attacker in the following equations.

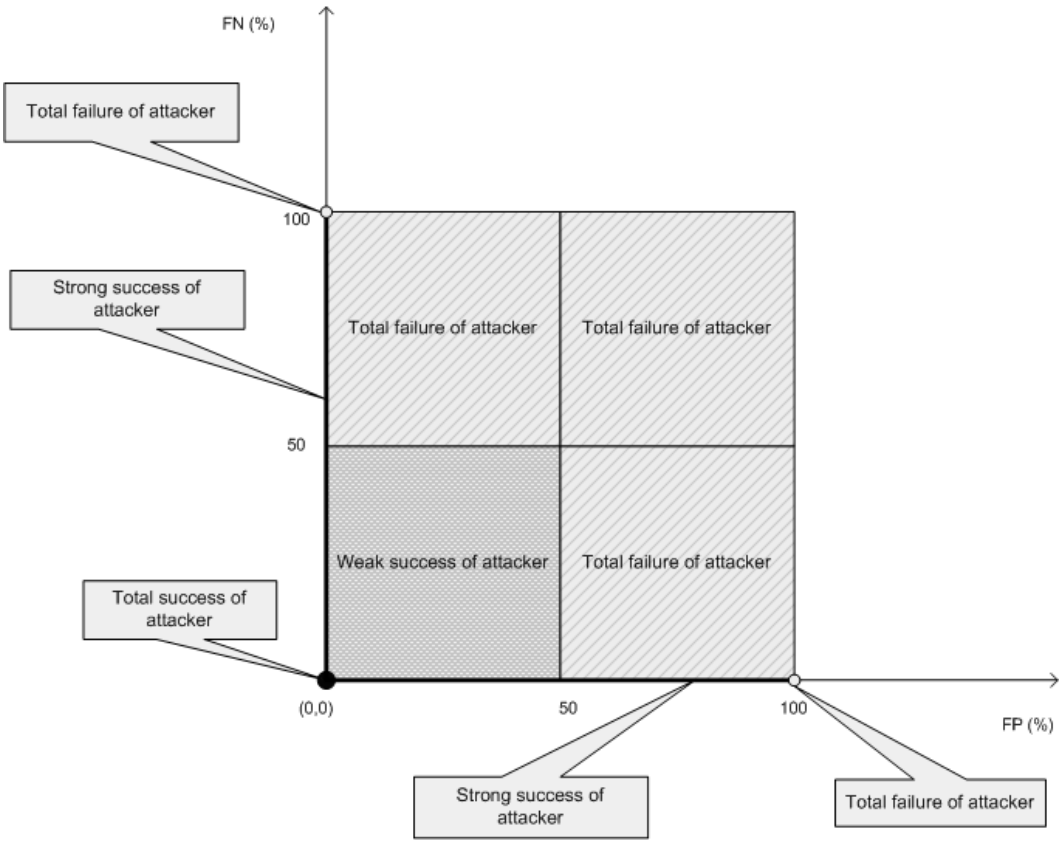


Figure 2.1: Judgement rules

$$max(FN, FP) \geq 50\% \rightarrow \text{total failure of attacker}$$
$$(\text{total success of defender}) \tag{2.3}$$

$$((0\% < FN < 50\%) \cap (0\% < FP < 50\%)) \rightarrow \text{weak success of attacker}$$
$$(\text{weak failure of defender}) \tag{2.4}$$

$$((FN = 0\%) \cap (0\% < FP < 100\%)) \rightarrow \text{strong success of attacker}$$
$$(\text{strong failure of defender}) \tag{2.5}$$

$$((FP = 0\%) \cap (0\% < FN < 100\%)) \rightarrow \text{strong success of attacker} \\ (\text{strong failure of defender}) \tag{2.6}$$

$$((FN = 0\%) \cap (FP = 0\%)) \rightarrow \text{total success of attacker} \\ (\text{total failure of defender}) \tag{2.7}$$

2.5 Discussion

In the discussion section, we discuss each of the four designers' limitations on attacker. Then the dangers of AUDW are also discussed. Lastly, we discuss issues about judgement rules.

2.5.1 Designers' Limitation 1 on Attacker

The reason behind designers' limitation 1 on attacker is: security society people generally believe that it is impossible to keep a security algorithm secret [21]. So the security must completely depend on the secret key [1]. This principle is called Kerckhoffs' principle which is the foundation of cryptography. Kalker [21] and Furon [17] suggest applying Kerckhoffs' principle in watermark design too. The CT watermark algorithm obeys this principle.

Then one question arises: what can an AUDW attacker learn from the CT algorithm? It is very likely that the attacker may notice that the CT algorithm will use many codes related to node building and node linking. Although node building and node linking codes is often used in object-oriented programming, it may still not be very stealthy in some cases. The relatively high frequencies (e.g.in cycle graph) of using nodes building and linking codes to build the CT watermark might still be 'abnormal'. That 'abnormality'

can then help attackers to distinguish the CT watermarked programs from unwatermarked programs. Additionally, the attackers can also assume that the opcodes used to build the CT watermark are those opcodes which can construct and link nodes. (Actually, in [11], the Java opcodes and their distributions are already given.) This information can be useful in the feature selection phase of our attack. However, we decide not to give the attacker in our experiment this information. Thus we can find how well SPSS can select features in its discriminant analysis.

Applying Kerckhoffs' principle in watermark design is doubtful. The problem comes from the fact that some techniques we used to defend against attackers can also be used by malicious adversaries. For example, an obfuscating method used to protect the CT watermark from detecting can also be used by adversaries. Adversaries can use that method to escape the detection of software plagiarism. On the other hand, techniques used in detection of software plagiarism can also be used in detection of software watermark by adversaries. So at least one type of adversary will win.

Although Kerckhoffs' principle is appropriate in cryptography, it is less appropriate in watermark design. Adversaries who attack the password put in enormous effort because the benefit of breaking passwords can be enormous, too. However, Adversaries who attack the watermark will not extend more effort beyond developing their own similar program. Similar program here means the program can perform the similar functions as the program protected by the watermark. Otherwise attackers who break the watermark can gain no economic benefits. So adversaries who attack watermark should be "weaker" than adversaries who break passwords. Another fact is that the effort paid to understand the watermark algorithm can be a large part of the effort paid to attack the watermark. As a result, Kerckhoffs' principle may be not quite suitable for watermark design consideration.

2.5.2 Designers' Limitation 2 on Attacker

Designers' limitation 2 on attacker gives the restriction that an attacker can not access the source code of the target watermarked program. Normally an attacker can seldom

access the source code of the target program in the real world. Since the Java class files can be easily disassembled into opcodes, we reason those opcodes and their sequences ($k-gram$) used as our features are appropriate.

2.5.3 Designers' Limitation 3 on Attacker

Designers' limitation 3 on attacker restricts the size of the CT watermarked program. That is, it will make it difficult for attackers to read the CT watermarked program manually. If the attacker must pay a significant effort, the security goal of the CT algorithm is already somehow achieved.

However, we should take care with this limitation. Some types of "manual" attacks are not quite "difficult" for attackers. For example, attackers may manually read the constant pool with ease while reading the entire codes of that program line by line is difficult. Another possible scenario is suggested in [11]. The attacker may develop an automated tool which can reveal a small part of suspicious codes related to watermark construction. Then he can check those suspicious codes manually.

In addition, we can not quantitatively define how large a file has to be to make it "difficult" for an attacker to read it "manually". The skill of reading code manually can vary much from attacker to attacker. What is more, the programmer can make the code more difficult to be read manually according to Roedy's paper [18].

But we can argue this limitation according the generally believed fact that programs are "Easier written than read" [35]. That is, the effort on reading (and understanding)the entire program line by line often will be more than the effort of writing the same program. The effort can be measured by person months. According to Norman et al. [27], "because staff costs often dominate overall project cost, the term 'cost estimation' and 'effort estimation' are sometimes used interchangeably". So we can make an assumption that the cost to read a program (manual attack cost) will also be more than the cost to write the same program (program developing cost). From an economic viewpoint, those attacking for business reasons will choose the lower cost between the manual attack cost

and the program developing cost. Since the program developing cost is often lower than the manual attack cost, attackers will not choose manual attack. That means we need not consider manual attack in most cases.

So when performing our AUDW, we can not exclude those relatively "small" programs. However, the success of detecting a watermarked program is related to the size of the target watermarked file. So we need to limit the file size to keep the CT watermark stealthy.

2.5.4 Designers' Limitation 4 on Attacker

Designers' limitation 4 on attacker excludes a certain type of attack based on dynamic analysis. It is not relevant to our attack which is based on static analysis.

2.5.5 The Danger of the Detection of Watermark Attack

Attackers can use the AUDW for two purposes:

- ***Attackers' purpose 1***: when attackers can use their $\mathfrak{detect}$ function to decide whether a unlabeled program is CT watermarked, they can collect programs judged as CT watermarked by their $\mathfrak{detect}$ function. The collected CT watermarked programs can be used for further attacks.

- ***Attackers' purpose 2***: when attackers can use their $\mathfrak{detect}$ function to decide whether an unlabeled program is CT watermarked, they can avoid pirating the programs watermarked by the CT watermark algorithm to avoid being caught.

One example of the dangers of AUDW is that if attackers can detect a watermark with $(FP = 0\%) \cap (0\% < FN < 100\%)$, then they can obtain more than one copy of watermarked programs p_w. So a collusion attack can be applied. A collusion attack relies on owning more than one copy of watermarked programs p_w [12]. In addition, the collusion attack can be more dangerous when more and more copies of watermarked programs are collected by the unauthorized $\mathfrak{detect}$ function.

2.5.6 The Judgement Rule

Equation (2.3) means that the detect function will do no better than a random guess which will have $max(FN, FP) = 50\%$. So the detect function is useless. We define such a detect function as the total failure of attackers.

Equation (2.4) means that the detect function can do better than a random guess. So the detect function will somehow reveal the existence of the watermark. However, this is risky for attackers because sometimes they will judge a watermarked program as an unwatermarked program. So we define such a detect function as a weak success for attackers.

Equation (2.5) means that the detect function can detect a watermarked program without judging an unwatermarked program as a CT watermarked program. The detect function will be useful for attackers if their purpose is the ***Attackers' purpose 1*** which was discussed in Section 2.5.5. It is because the collected watermarked programs by the attacker will not mix with unwatermarked programs. So we define such a detect function as a strong success for attackers.

Equation (2.6) means that the detect function can detect a watermarked program without judging a watermarked program as an unwatermarked program. The detect function will be useful for attackers if their purpose is the ***Attackers' purpose 2*** which was discussed in Section 2.5.5. It is because the attacker will not risk pirating a CT watermarked programs. So we define such a detect function as a strong success for attackers.

Equation (2.7) means that the detect function will not make any mistakes, so it a full success for attackers.

3

Pattern Classification Theory

3.1 Introduction

In this chapter, we focus on pattern classification theory and the choices involved in pattern classification procedure. How those different choices will change the accuracy of classification results are also discussed. In the next chapter, we will show that our experiment design obeys the theory discussed in this chapter.

3.2 Overview of Pattern Classification

Pattern classification is one method which can be used for the detecting watermarks. In the procedure of watermark detection, unknown programs are classified into two cate-

gories: watermarked programs C_w and unwatermarked programs C_u. When the decision is based on some patterns of programs, it is an application of pattern classification. Usually, pattern classification system will include steps such as *data collection*, *feature choice*, *model choice*, *training*, *evaluation*[16]. Steps can be repeated more than once. Our experiment design will basically follow those steps. We will discuss them in the following subsections.

3.2.1 Data Collection

The most important thing in data collection is to ensure the collected sample set can represent the sample set to be collected [40]. Otherwise we must reason why and how the samples collected are different from the samples to be collected [40]. Webb [40] also discussed that other aspects should be considered in data collection. Those aspects include the total sample *size*, the *ratio* of sample numbers between C_w and C_u, *cost* related to data collection and the randomly sampling *rule*. Our choices are as follows.

- ***sample size***: The maximum size (number of samples) of our collected sample set is 234. It is because that we decide to only use the same 234 samples collected by Collberg in our experiment. We will not optimize our sample set as presented in Section 3.3.1. It will be one of the reasons for lowering the accuracy of our classifier.

- ***the ratio between*** C_w ***and*** C_u: In the real world, $C_w \ll C_u$. In our experiment, we set our ratio of C_w to C_u above 1:150 for training purposes. The training procedure will be discussed in Section 3.2.4

- ***cost related to data collection***: In our experiment, the main cost related to data collection is in embedding watermark manually.

- ***randomly sampling rule***: We are using the sample set from the research group of Prof. Collberg. We choose to trust their randomly sampling rule which is used to collect the sample set.

3.2.2 Feature Choice

Feature choice includes two-level choices: level one choice involves features suitable for representing softwares while level two choice has features for data reduction purposes. Both of these considerations are described, and our reasons for choosing them, below.

Choosing Features Suitable for Representing Softwares

We choose opcode level *k-gram*s as features to present the patterns of software programs because it is a general technique for detecting the similarities between programs [4, 25]. We only consider the situation that $k \leq 3$ in our experiment. We keep k a small number because the number of features will increase dramatically when k increases. The so-called *curse of dimensions* problem will be discussed in Section 3.3. Features can also be based on bytecode sequences or other software metrics. We choose *k-gram*s as our features because they are resilient to semantic-preserving transformations [25].

We retrieve the pattern of a program by measuring the features ($k - grams$) of that program. The details about how to measure the $k - grams$ to form the pattern of a program are shown in Figure 4.1.

Choosing features for data reduction purposes

For data reduction purposes, we will select only a subset of *k-gram*s in our pattern classification. The need to reduce the number of features for two reasons [16, 40]:

1. Reducing the time and space used in statistical analysis.

2. Increasing the accuracy of statistical analysis.

We will discuss the second reason more in Section 3.3.

3.2.3 Model Choice

Our model choice (classifier choice) is Fisher's linear discriminant classifier. A classifier is a "rule to assign a class (or doubt or outlier) to new examples" [3]. A classifier takes an input

example (a program), and gives a classification result as the output. The classification result is given by applying a classification algorithm to some measurements of the features of that program. We name the classifier after the classification algorithm used in this thesis.

According to Raudys [32], there exist more than two hundred pattern classification algorithms and six important algorithm are described. These six important algorithm have demonstrated practical uses. Among them, Fisher's linear discriminant algorithm is the most popular [32]. Besides, it often gives good results [32]. So very likely, an attacker would be likely to try Fisher's linear discriminant classifier first, especially when he/she does not have other information to determine which classifier is better.

3.2.4 Training

A training procedure adjusts the coefficients of the classification function to fit those "known programs". "known programs" means that the attacker knows whether the programs are watermarked or not.

"Known programs" are those programs whether watermarked or not are known to the attacker. After training, we can use the classification function with adjusted coefficients to classify new programs or unknown programs.

3.2.5 Evaluation

Evaluation estimates the "real" misclassification rate by additional experiments. Four main methods are often used [32].

- The *resubstitution* method: All training samples are reused in evaluation
- *The hold-out* method: The sample set is divided into two parts. One part is for training and another part is for evaluation purpose. We adopt this method for our experiments.
- *The cross-validation* method: Suppose we have n number of samples, each time we

select k samples as the training set and the remaining $n-k$ samples as the test set. When the k samples should not be repeated, then we can result in $\binom{n}{k}$ choices. Each choice will have an error rate. Finally the average error rate of all the $\binom{n}{k}$ choices will be computed and used in evaluation.

- *The bootstrap* method: In bootstrap, resampling samples from the training set r times forms r bootstrap sample sets. The error rate of each bootstrap sample set is compared with the error rate of the whole training set. Then we can use the differences of error rate for evaluation purpose.

3.3 Discussion

In this section, we will discuss how to optimize the pattern classification activities. In practise and in our experiment, an attacker may not be able to optimize each of his/her pattern classification activities. Then the error rate of attacker's `detect` function will increase.

3.3.1 Data Collection

To determine the appropriate sample size, we must consider several factors. Those factors include dimension d, the cost of classification the desired performance, the complexity of the classification rule and the asymptotic probability of misclassification. However, it is very difficult to theoretically calculate the relationship between the finite sample size and the performance of a certain classifier performance. So we just adopt the guidance of "having 5-10 times more samples per class than feature measurements"[40].

3.3.2 Feature Choice

Data reduction is important because of the *curse of dimensions*. The *Curse of dimensions* means that the complexity of high dimensions greatly increase the difficulty in developing an optimized discriminant function [16].

The dimension used in pattern classification can affect both of the feature extractor and classifier [16]. An high dimension will increase "the cost and complexity of both the feature extractor and classifier" [16]. An high dimension can also affect PMC as described below:

In that if the training sample size is unlimited, adding new features will help to increase the accuracy of the classifier (or reduce PMC) [16]. At least, adding new features will not increase PMC. So the higher the dimension, the lower the PMC.

However, if the sample number is fixed, according to [32], a *peak phenomenon* appears. With a *peak phenomenon*, the initial increase of feature numbers will reduce the possibility of misclassification (PMC). However, when the feature numbers exceed a certain number (d_{opt}), the increase of feature numbers will increase PMC too. In the case of using linear discriminant function, $d_{opt} = \frac{N}{2} - 1$ when N is the total number of features.

3.3.3 Model Choice

Fisher's discriminant function might not be the best function we can use. However, according to the *No Free Lunch Theorem* [16], no classifier or learning algorithm is always better than others without considering the nature of the task. The nature of the task includes those factors that can affect the PMC. According to Raudys [33], the PMC of all discriminant functions depend on the following factors :

- The classification rule type;
- Training sample size $|\mathcal{P}_t|$;
- Asymptotic probability of misclassification PMC_∞

$$PMC_\infty = \lim_{|\mathcal{P}_t|\to\infty} EPMC(\text{expect PMC when training size is}|\mathcal{P}_t|)$$

  ;

- Dimensionality d.

Raudys also gives the analytical formulas to calculate the PMC of four popular linear discriminant functions (including Fisher's discriminant function which is called "standard linear DF" in that article). So we can obtain the theocratical PMC of Fisher's discriminant function when we know the PMC_∞, $|\mathcal{P}_t|$, and d. If we can calculate the $EPMC$, then we know the theoretical PMC which is independent from those samples in the training set. So we can overcome one of the restrictions of AUDW: our experiment result is valid only for our collected sample set $\mathcal{P}_{E_0}$. Nevertheless, the computation is complex. So in practice Raudys [33] uses the average error rate of 10-100 runs to estimate the $EPMC$, where each run should randomly draw $|\mathcal{P}_t|$ samples from the sample set.

Such a calculation needs $|\mathcal{P}_t|$ to be relatively large. However, currently we do not have many CT watermarked programs for training. Since we can only embed watermarked programs manually, the time spent on preparing a large number of CT watermarked programs will be too much for our experiment. So calculating the $EPMC$ is beyond the scope of our experiment.

Currently we can not estimate well which classification algorithm is better. There are no published reports comparing classification algorithm on the accuracy of classifying watermark and unwatermarked programs. We can only select the classification algorithm in our experiment by our experience. In our experiment, we select Fisher's discriminant function as our classification algorithm. If we consider only the accuracy of classifiers, Fisher's discriminant function classifier might not be the best choice. According to [33], when $|\mathcal{P}_t|$ is small, it is better to choose simpler classification rules or reduce d. Some algorithms like Euclidean discriminant function may work better. However, it is easier to select features by using Fisher's discriminant function. The selection procedure can be automatically done in SPSS. That is why we finally choose Fisher's discriminant function.

3.3.4 Training Procedure

In the training procedure, we should be aware that our classifier should not overfit the training set. Overfitting means our classifier is too complex so that it works excellently

on training set but poorly on test set [16]. To avoid overfitting, we need to control d.

3.3.5 Fisher's Discriminant Pattern Classifier

Fisher's discriminant pattern classifier should be based some assumptions. SPSS, the tool we use to do the statistical analysis work, also uses those assumptions. However, in the real world, those assumptions often are not valid. We will discuss the assumptions and the effect of violating them in Chapter 4.

4

Experiment Design

4.1 Introduction

For the most part, our experiment design is based on the classification design cycle as we discussed in Chapter 3. We also adopt the experiment design techniques presented in [19].

In this experiment design, we introduce the two roles of attacker and experimenter. The attacker is confined by the restrictions on attackers as defined in Chapter 2. His/her job is to develop a classifier (or `detect` function) on the samples provided by the experimenter. The experimenter's job is to distribute samples to the attacker and evaluate the classifier developed by the attacker. The relationship between the experimenter and the attacker will be discussed in Section 4.5.

As discussed in Chapter 2, the error rate (FN and FP) of the attacker's $\mathfrak{detect}$ function determines the winner of the match between the watermark designer and the attacker. So we make some hypotheses on factors that can affect the error rate of the attackers' $\mathfrak{detect}$ function. We design one main experiment set and two additional experiment sets to verify our hypotheses.

4.2 Experiments

All the experiments we made in this thesis form our experiment set.

Definition (Experiment Set)

The experiment set $\mathcal{X}$ is an ordered set of **run**s. A run is an experiment which runs with a certain combination of experiment parameters [19], and $\mathbf{run}_r$ denotes the $r-th$ run in our case [15].

$$\mathcal{X} = \{run_1, run_2, \ldots, run_r, \ldots, run_{14}\}$$

and $\mathcal{X}_A$ is a subset of $\mathcal{X}$ which is our main experiment set.

$$\mathcal{X}_A = \{run_1, run_2, \ldots, run_9\}$$

$\mathcal{X}_B$ and $\mathcal{X}_C$ are also subsets of $\mathcal{X}$ which is our additional experiment sets.

$$\mathcal{X}_B = \{run_{13}, run_{14}\}$$

$$\mathcal{X}_C = \{run_{10}, run_{11}, run_{12}\}$$

Both of $\mathcal{X}_A$ and $\mathcal{X}_C$ are based on pattern classification. We use $\mathcal{X}_A$ and $\mathcal{X}_C$ to simulate the AUDW of attackers. The goals of $\mathcal{X}_A$ and $\mathcal{X}_C$ are to find factors which can significantly affect the accuracy of attacker's $\mathfrak{detect}$ function. Details of $\mathcal{X}_A$ and $\mathcal{X}_C$ will be discussed in Sections 4.8 and 4.9 respectively.

$\mathcal{X}_B$ is a simple experiment with only one run. With it, we explore the change of watermark size ratio caused by the different selection of watermark embedding parameters.

We will give the experiment flowchart of $r-th\ run$ in Section 4.7 ($1 \leq r \leq 12$). The 13-th and 14th *run*s ($\mathcal{X}_B$) are simple, so the flowchart of them is not need.

4.3 Sample Sets

The sources of our experiment sample sets are from the Java programs collected by Collberg's group from the internet. We assume none of those Collberg's samples are watermarked.

Definition (Original Experiment Sample Set)

Let the $\mathcal{P}_{E_0}$ be the finite, ordered subset of $\mathcal{P}_u$ which is selected for our experiment:

$$\mathcal{P}_{E_0} = \{p_1, p_2, \ldots, p_i, \ldots p_{234}\}$$

where p_i is the i-th program in $\mathcal{P}_{E_0}$ (Table A.2 has the name and size of the element programs in $\mathcal{P}_{E_0}$.)

Our experiment requires building some CT watermarked programs. The watermarks used for those programs for our experiment are listed in Table A.4.

Definition (Watermark Set for Experiment)

Let watermark set for experiment $\mathcal{W}_E$ be the subset of $\mathcal{W}$ which will be used in our experiment. (The elements of $\mathcal{W}_E$ are listed in Table A.4.) We denote $\mathcal{W}_E^r$ as the subset of $\mathcal{W}_E$ which is used in $r-th\ run$.

The combinations of CT watermark embedding parameters in Sandmark should also be considered in our experiment. For our experiment, we focus on two options of embedding parameters: *numeric watermark*, *use cycle graph*. *Numeric watermark* means that Sandmark will only accept a numeric number as watermark [7]. Collberg claims that by using only the numeric number as watermark, the watermark algorithm will be

parameters selection	Numeric watermark	Use Cycle graph	Other fixed parameters $*$
m_1	0	1	$**$
m_2	1	0	$**$

"1" means select, "0" means not select
$*$ The fixed parameters list in Appendix A.2.4.
$**$ Using default value. Those default values are listed in Appendix A.2.4.

Table 4.1: Combinations of CT watermark embedding parameters in Sandmark

more efficient than using an arbitrary string watermark [7]. *Use cycle graph* means that Sandmark will build a more complex watermark graph to resist node splitting attacks [7]. When Sandmark embeds watermark by using *use cycle graph* option, it will replace each node in the watermark graph with a 3-node cycle [7]. In our experiment, we will only consider these two options. Other fixed watermark embedding parameters are kept in their default values. The default values of those fixed watermark embedding parameters are in Appendix A.2.4.

Definition (Embedding parameter combination Set for Experiment)

Let embedding parameter combination Set for Experiment $\mathcal{M}_E$ be the ordered subset of $\mathcal{M}$. $\mathcal{M}_E = \{m_1, m_2\}$. The combinations used for m_1, m_2 are listed in Table 4.1.

Definition (Experiment Sample Set)

Let $\mathcal{P}_E^r$ be the subset of $\mathcal{P}$ which is used for $r-th\ run$. We map $\mathcal{P}_E^r$ from $\mathcal{P}_{E_0}$ by the $\mathfrak{transform}$ function (for $r \leq 12$).:

$$\mathcal{P}_{E_0} \times \mathcal{W}_E \times \mathcal{K}_E \times \mathcal{M}_E \rightarrow \mathcal{P}_E^r$$

$$\mathfrak{transform}(p_{0,j}) = \begin{cases} p_j = \mathfrak{embed}(p_{0,j}, w_j, key_j, m_1) & if\ j \in \{1,5,15,16,18,30,50,54,60, \\ & 69,73,77,78,87,92,123,141,155, \\ & 197,207,234\} and\ (\ 10 \leq r \leq 12) \\ p_j = \mathfrak{embed}(p_{0,j}, w_{j+1000}, key_j, m_1) & if\ j \in \{1,5,15,16,18,30,50,54,60, \\ & 69,73,77,78,87,92,123,141,155, \\ & 197,207,234\}\ and\ (\ 1 \leq r \leq 9) \\ p_j = \emptyset & if\ j \in \{19,37,9,22,24,33,40,47,48, \\ & 88,94,100,103,114,119,125,138, \\ & 147,157,159,162,164,175,176,177, \\ & 178,207,229,230,129,152, \\ & 144,75,143,182\} \\ & and\ (\ 1 \leq r \leq 9) \\ p_j = p_{0,j} & otherwise \end{cases}$$

All watermarked programs in $\mathcal{P}_E^r$ are denoted as $\mathcal{P}_{w,E}^r$.

$$\mathcal{P}_{w,E}^r \ = \ \mathcal{P}_w \cap \mathcal{P}_E^r;$$

All unwatermarked programs in $\mathcal{P}_E^r$ are denoted as $\mathcal{P}_{u,E}^r$.

$$\mathcal{P}_{u,E}^r \ = \ \mathcal{P}_u \cap \mathcal{P}_E^r$$

In $r-th\ run$, $\mathcal{P}_E^r$ will be split into $\mathcal{P}_t^r$ and $\mathcal{P}_v^r$. $\mathcal{P}_t^r$ is the subset of $\mathcal{P}_E^r$ which is used by adversaries to build $classifier^r$ in the $r-th\ run$. $\mathcal{P}_v^r$ is the subset of $\mathcal{P}_E^r$ which is used by the experimenter to verify the accuracies of adversaries' $classifier_r$. $\mathcal{P}_t^r$ and $\mathcal{P}_v^r$ should be *mutually exclusive*:

$$\mathcal{P}_t^r \ \cap \ \mathcal{P}_v^r \ = \ \phi$$

The subsets of $\mathcal{P}_t^r$ and $\mathcal{P}_v^r$ are:

$$\mathcal{P}_{w,t}^r : \mathcal{P}_{w,t}^r = \mathcal{P}_w \cap \mathcal{P}_t^r$$

$$\mathcal{P}_{w,v}^r : \mathcal{P}_{w,v}^r = \mathcal{P}_w \cap \mathcal{P}_v^r$$

$$\mathcal{P}_{u,t}^r : \mathcal{P}_{u,t}^r = \mathcal{P}_u \cap \mathcal{P}_t^r$$

$$\mathcal{P}_{u,v}^r : \mathcal{P}_{u,v}^r = \mathcal{P}_u \cap \mathcal{P}_v^r$$

4.4 Pattern

As we mentioned in Chapter 1, a pattern consists of the measurements of a set of characteristic features of an object [40]. In our experiment, the characteristic features are $k-grams$.

Definition (Experiment Gram Set)

Let the gram set for $r-th\ run$ be $\mathcal{G}^r$. $\mathcal{G}^r = \{k-gram_1, k-gram_2, \ldots, k-gram_i, \ldots, k-gram_n\}$, where k is the length of the $k-gram$, n is the number of $k-grams$ in $\mathcal{G}^r$. For a particular $r-th\ run$, we always use the same k. So $\mathcal{G}^r$ is a subset of $\mathcal{A}^k$.

$$\mathcal{A}^k = \underbrace{\mathcal{A} \times \ldots \times \mathcal{A}}_{n \text{ times}}$$

where $\mathcal{A}$ is defined as the ordered set of Java opcodes, $\mathcal{A} = \{op_1, op_2, \ldots, op_j, \ldots, op_{206}\}$; op_j is the j-th opcode in $\mathcal{A}$; the elements of $\mathcal{A}$ are shown in Table A.1.

In $r-th\ run$, we retrieve a pattern vector $\overrightarrow{p}^r$ from a p by a measure procedure. The measure procedure is based on $\mathcal{G}^r$. The pattern vectors retrieved from $\mathcal{P}_E^r$ are used by attackers to train his $classifier_r$ and used by experimenter to evaluate attackers' $classifier_r$.

Definition (Pattern Set)

Let $\mathcal{T}_*^r$ be the set of pattern vectors retrieved from $\mathcal{P}_*^r$ in $r-th\ run$. $\mathcal{T}_*^r$ relates to $\mathcal{P}_*^r$ for the $r-th\ run$ under the *measure* procedure described in Figure 4.1; the *measure* procedure will take a $p \in \mathcal{P}_*^r$ and the $\mathcal{G}^r$ as input and return a pattern vector $\overrightarrow{p} \in \mathcal{T}_*^r$ as output; e.g. $\mathcal{T}_{w,t}^r$ relate to $\mathcal{P}_{w,t}^r$ for the $r-th$ run.

Figure 4.1 shows the measure procedure in $r-th\ run$ for a arbitrary program whose jar file is called A.jar. The meaning of each step is as follows:

- *Step 1*: We obtain Java .class files for Java .jar file then disassemble Java class files into Java byte code.
- *Step 2*: We filter out the opcode list.
- *Step 3*: We sweep a window of size k on the opcode list to retrieve the $k-gram$ sequence of A.jar.
- *Step 4*: We count the frequencies of occurrence of all $k-grams$ belong to $\mathcal{G}^r$ upon the $k-gram$ sequence of A.jar retrieved in *Step 3*. Then we obtain a frequency vector $\overrightarrow{f}^r$ of A.jar for $r-th\ run$. $\overrightarrow{f}^r = \{f_1, \ldots, f_n\}$, where n is the number of $k-grams$ in $\mathcal{G}^r$.
- *Step 5*: We retrieve the pattern vector $\overrightarrow{p}^r$ from $\overrightarrow{f}^r$. $\overrightarrow{p}^r = \{f_i' \in \overrightarrow{p}^r, f_i \in \overrightarrow{f}^r : f_i' = \frac{f_i}{\sum_{i=1}^{n} f_i}\}$

4.5 Two Roles

We define two roles operate in our experiment design: the attacker and the experimenter. As a restriction in Section 2.3, an attacker can not access the official **embed** and **extract** functions of the CT watermark algorithm. Also an attacker can not evaluate his/her classifier. So we need to define a role as experimenter to prepare the $\mathcal{P}_E^r$ for $r-th\ run$ and evaluate the classifier of attacker.

Figure 4.2 shows the relationship between them in the $r-th\ run$. We denote $attacker^r$ as the attacker in the $r-th\ run$. In the $r-th\ run$, the experimenter distributes the

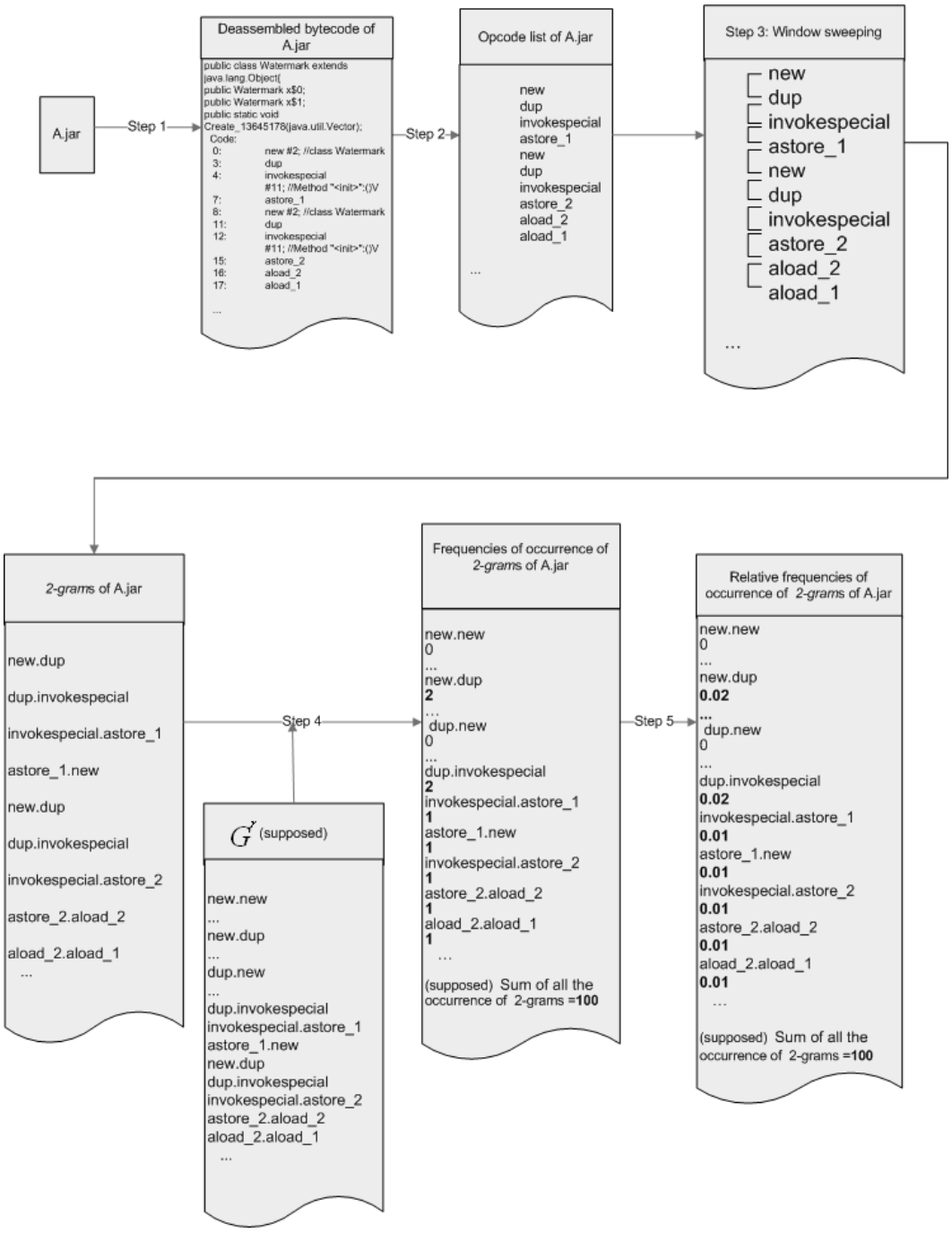
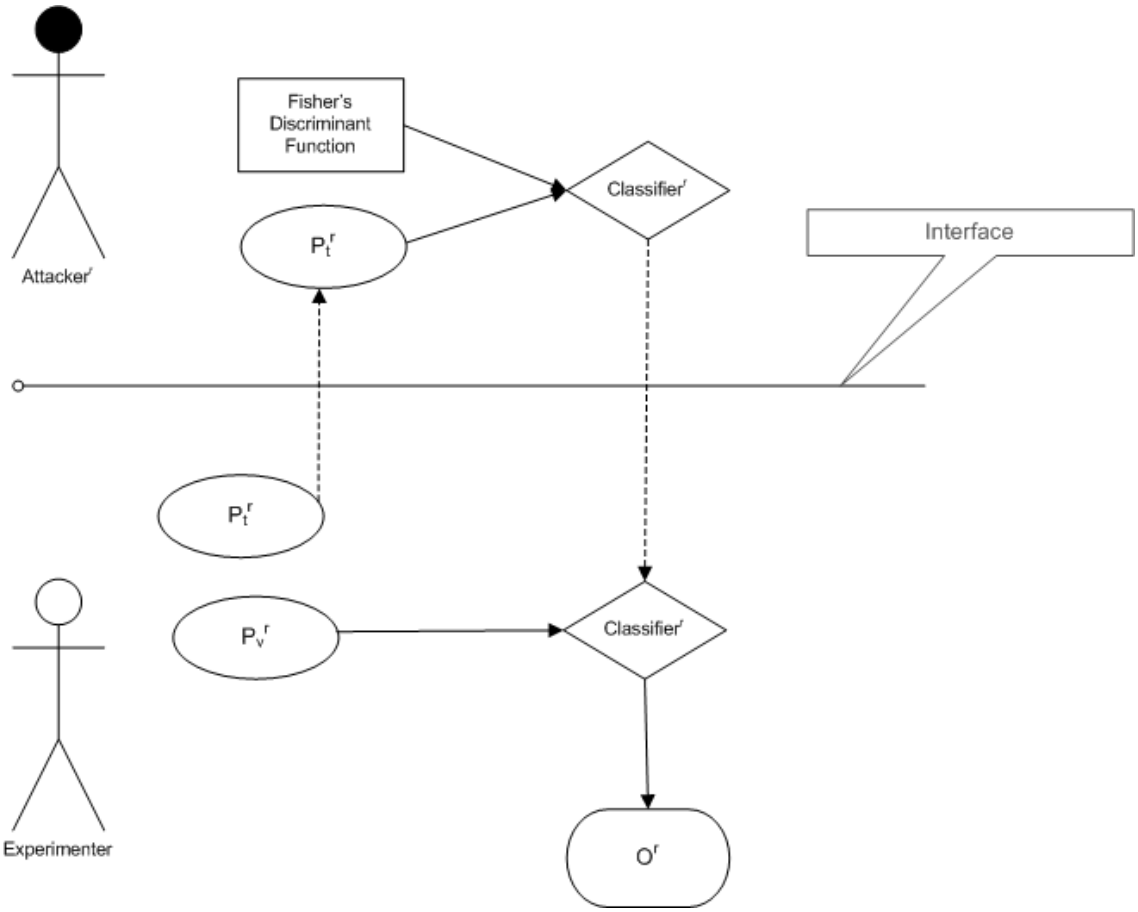


Figure 4.1: Measuring Procedure

$\mathcal{P}_t^r$ to the $attacker^r$, then the $attacker^r$ trains his/her $classifier^r$ with $\mathcal{P}_t^r$ and Fisher's discriminant function. The trained $classifier^r$ will be evaluated by the experimenter with $\mathcal{P}_v^r$. The experimenter uses FN^r and FP^r to evaluate the accuracy of $classifier^r$. We



Note: dotted line means exchanges between the Attacker and the Experimenter

Figure 4.2: The Relationship Between the Attacker and the Experimenter in $r-th\ run$

denote the output of $r-th\ run$ as $\mathcal{O}^r$.

$$\mathcal{O}^r \ = \ \{FN^r, FP^r\}$$

4.6 Hypotheses

In our experiment, we try to explore the factors which will significantly affect the accuracy of an attacker's classifier. Since no surveyed paper can report which factor can significantly affect the accuracy of an attacker's classifier, we keep hypotheses and verify them by our experiment.

Our main hypothesis relates to the levels of watermark size ratios of $\mathcal{P}^r_{w,t}$ and $\mathcal{P}^r_{w,v}$. The watermark size ratio is defined as follows.

Levels	definitions
QTL_1	$\forall q \in \mathcal{Q}^r_{w,t}, 0 < q \leq 0.2$
QTL_2	$\forall q \in \mathcal{Q}^r_{w,t}, 0.2 < q \leq 2.0$
QTL_3	$\forall q \in \mathcal{Q}^r_{w,t}, 2.0 < q$
QVL_1	$\forall q \in \mathcal{Q}^r_{w,v}, 0 < q \leq 0.2$
QVL_2	$\forall q \in \mathcal{Q}^r_{w,v}, 0.2 < q \leq 2.0$
QVL_3	$\forall q \in \mathcal{Q}^r_{w,v}, 2.0 < q$

Table 4.2: The levels of watermark size ratio of $\mathcal{P}_{w,t}$ and $\mathcal{P}_{w,v}$

Definition (watermark size ratio)

Let q be the watermark size ratio of a watermarked program, $\mathcal{Q}^r_{w,*}$ is the set of watermark size ratios of programs in $\mathcal{P}^r_{w,*}$, $\mathcal{Q}^r_{w,*} \subseteq \mathbb{R}$, and the $\mathfrak{ratio}$ function maps $\mathcal{Q}^r_{w,*}$ from $\mathcal{P}^r_{w,*}$:

$$\mathcal{Q}^r_{w,*} = \{q \in \mathcal{Q}^r_{w,*}, p_w \in \mathcal{P}^r_{w,*} : q \ = \ \mathfrak{ratio}(p_w)\}$$

The $\mathfrak{ratio}$ function is defined as:

$$\mathfrak{ratio} : \mathcal{P}_w \rightarrow \mathbb{R}$$

$$\mathfrak{ratio}(p_w) \ = \ \{p_w \in \mathcal{P}_w : \frac{|p_w| - |p_u|}{|p_u|}\}$$

Where $p_w \ = \ embed(p_u, w, key, m)$; w is the watermark to be embedded; key is the input sequence used to embed watermark, m is the combination of parameters used in watermark embedding. $|p_w|$ and $|p_u|$ are the size of p_w and p_u respectively, the size is measured by the sum of all the frequencies of the occurrence of all *op*s in a program p. e.g.a program $p \equiv \langle ADD, SUB, ADD, ADD, SUB, AASTORE \rangle$, then $|p| \ = \ 6$

Then we denote the levels of watermark size ratios of $\mathcal{P}^r_{w,t}$ as QTL^r: $QTL^r \in \{QTL_1, QTL_2, QTL_3\}$, where QTL_1, QTL_2, QTL_3 are defined in Table 4.2.

Similarly, we denote the levels of watermark size ratios of $\mathcal{P}^r_{w,v}$ as QVL^r: $QVL^r \in \{QVL_1, QVL_2, QVL_3\}$, where QVL_1, QVL_2, QVL_3 are defined in Table 4.2.

Our main hypothesis of the experiment is **Hypothesis One**: While all other experiment parameters are fixed, FN^r and/or FP^r of attacker's $classifier^r$ will decrease

when QTL^r and/or QVL^r increase. On the other hand, FN^r and/or FP^r of attacker's $classifier^r$ will increase when QTL^r and/or QVL^r decrease.

The basis of **Hypothesis One** comes from a well known fact in media watermark techniques: The error rate of media watermark detection will increase when the size of the embedded watermark decreases for a fixed media file size [38, 5].

Although software watermark algorithms are quite different from the media watermark algorithms, they share many of the same information hiding problems. i.e. in media watermarking, watermark designers should control their watermark size small enough compare to the size of the picture to be watermarked. So the watermark will be imperceivable to human eyes [38, 5]. In software watermarking, we believe that we should also keep the size of the watermark building codes small compare to the size of program to be watermarked. Thus statistical analyzing methods used by attackers can not detect the watermark.

We test **Hypothesis One** using $\mathcal{X}_A$. The details to implement $\mathcal{X}_A$ will be discussed in Section 4.8.

If **Hypothesis One** is true, then we need to consider the variables which can change the *watermark size ratio*. According to Collberg and Townsend [7], in Sandmark, when we embed watermark using the "*numeric watermark*" option, the watermark embedded will be more efficient. When we embed watermark using the "*cycle graph*" option, each node of the watermark graph will be changed into a 3-cycle. So we assume using the "cycle graph" option will use more codes to build the same watermark than using the "numeric watermark" option.

So our **Hypothesis Two** states that for the same watermark and same program, embedding a watermark employing m_2 parameter combinations (using the "*numeric watermark*" option) will cause lower *water mark size ratio* than employing m_1 parameter combinations (using the "*cycle graph*" option) in Sandmark.

To verify our **Hypothesis Two**, we design a simple one factor experiment $\mathcal{X}_B$ to check it. The details of how to implement $\mathcal{X}_B$ will be in Section 4.9.

The idea of our third hypothesis is from Christian Collberg and Clark Thomborson [11]. They believe that the selection of the length of $k-grams$ will affect the accuracy

of attacker's $\mathfrak{detect}$ function. So we want to check the effect of the length of $k-gram$ on the accuracy of attacker's $\mathfrak{detect}$ function.

Our last hypothesis is **Hypothesis Three** where we posit that the selection of k (length of $k-grams$) in the feature selection phase of attack will change the FN^r and/or FP^r of that attacker's $\mathfrak{detect}$ function while all other experiment parameters are fixed. To verify our **Hypothesis Three**, we design a simple experiment $\mathcal{X}_C$ to check it. The implementation details of $\mathcal{X}_C$ will be in Section 4.10.

4.7 Experiment Flowchart

We design our experiment mainly according to the design cycle described in Chapter 3. The design cycle consists of following activities: *data collection*, *feature choice*, *model choice*, *training* and *evaluation*. In the design cycle described in *Pattern Classification* [16], some activities will repeat to optimize the outputs of the experiment.

However, our experiment design is slightly different from the above design cycle. We assume that attackers can not repeat above the activities to achieve better results to simplify our experiment. Each activity is executed only once.

Figure 4.3 is the flowchart of $r-th\ run$ in our experiment ($1 \leq r \leq 12$). Each $r-th$ run consists of six steps ($1 \leq r \leq 12$). The six steps sequentially implement the activates of *data collection*, *feature choice*, *model choice*, *training* and *evaluation*. Details are as follows:

- *Step 1*: The experimenter builds $\mathcal{P}_E^r$ from $\mathcal{P}_{E_0}$ by the $\mathfrak{transform}$ function.
- *Step 2*: The experimenter builds $\mathcal{P}_{w,t}^r$, $\mathcal{P}_{u,t}^r$, $\mathcal{P}_{w,v}^r$, $\mathcal{P}_{u,v}^r$ from programs in $\mathcal{P}_E^r$.
- *Step 3*: The experimenter distributes $\mathcal{P}_{w,t}^r$ and $\mathcal{P}_{u,t}^r$ and $\mathcal{G}^r$ to $attacker^r$ and $attacker^r$ uses them to retrieve the pattern set $\mathcal{T}_t^r$.
- *Step 4*: The $attacker^r$ builds and trains his $\mathfrak{detect}^r$ function (or $classifier^r$) by $\mathcal{T}_t^r$ and using Fisher's linear discriminant function.

- *Step 5*: The experimenter uses $\mathcal{P}^r_{w,v}$, $\mathcal{P}^r_{u,v}$ and $\mathcal{G}^r$ to retrieve the pattern set $\mathcal{T}^r_v$.

- *Step 6*: The experimenter evaluates $attacker^r$'s $\mathfrak{detect}^r$ function.

- *Step 7*: The experimenter obtain the FN^r and FP^r of $attacker^r$'s $\mathfrak{detect}^r$ function as $\mathcal{O}^r$.

Step 1 and 2 implement the *data collection* activity in the pattern classification design cycle. Step 3 implements the *feature selection* and *model choice* activities. Step 4 implements the *training* activity. Steps 5, 6 and 7 implement the evaluation activity.

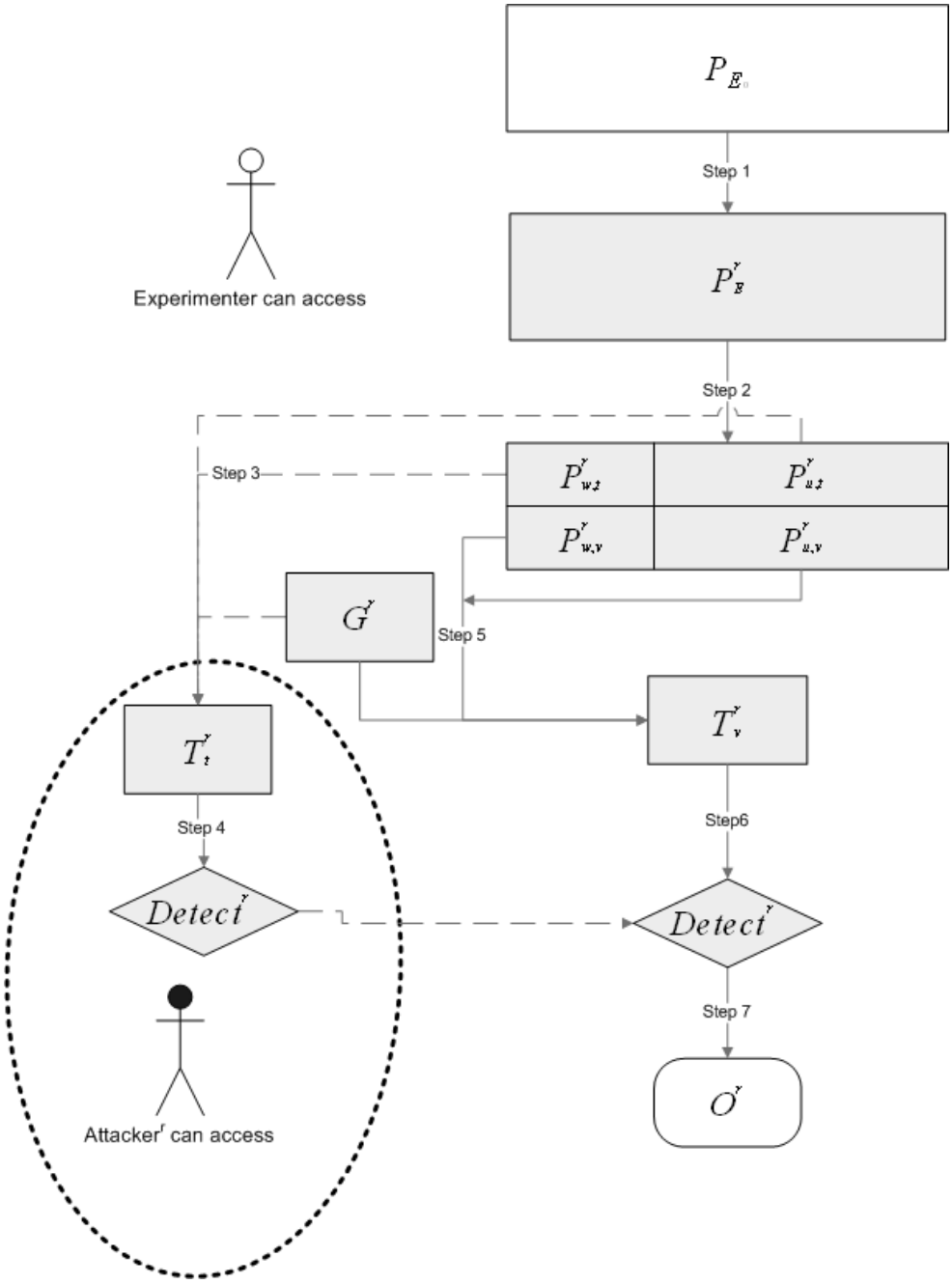
In Figure 4.3, the $attacker^r$ can access items only inside the dotted oval area. On the other hand, the experimenter can access items only outside the dotted oval area.

4.8 Design for Experiment Set A

We adopt the *two-factor full factorial design without replications* [19] as our experiment design for experiment set A $\mathcal{X}_A$. We choose this design because we want to test combinations of all levels of those two factors while each combination runs only once[19]. The two factors we will explore in our experiment are called primary factors [19] and those we selected for $\mathcal{X}_A$ are QTL^r and QVL^r.

Table 4.3 shows the experiment design for experiment set A. Table 4.4 shows the elements of $\mathcal{P}^r_{w,t}$ for experiment set A. Table 4.5 shows the elements of $\mathcal{P}^r_{w,v}$ for Experiment set A. For all *run*s of $\mathcal{X}_A$:

$$\mathcal{P}^r_{u,v} = \{p_i : p_i \in \mathcal{P}^r_E, i \in \{11, 14, 25, 35, 38, 49, 85, 93, 112, 133, 105, 148, 156, 163, 164, 174, 137, 189, 191, 196\}\}$$



(Notice: The fonts of Sets are slightly changed because the drawing software do not support fonts used by Latex)

Figure 4.3: Flowchart of $r-th\ run$ in Experiment ($1 \leq r \leq 12$)

For all $runs$ of $\mathcal{X}_A$:

$$\mathcal{P}^r_{u,t} = \{p_i : p_i \in \mathcal{P}^r_E, i \notin \{11, 14, 25, 35, 38, 49, 85, 93, 112, 133, 105, 148,$$
$$156, 163, 164, 174, 137, 189, 191, 196, 1, 5, 15, 16,$$
$$18, 30, 50, 54, 60, 69, 73, 77, 78, 87, 92, 123, 141, 155,$$
$$197, 207, 234\}\}$$

Levels	QTL_1	QTL_2	QTL_3
QVL_1	run_1	run_2	run_3
QVL_2	run_4	run_5	run_6
QVL_3	run_7	run_8	run_9

Table 4.3: experiment design for Experiment Set A

$\mathcal{P}^r_{w,t}$	Elements
$\mathcal{P}^r_{w,t}, r \in \{1,2,3\}$	$\{p_i : p_i \in \mathcal{P}^r_E, i = 73\}$
$\mathcal{P}^r_{w,t}, r \in \{4,5,6\}$	$\{p_i : p_i \in \mathcal{P}^r_E, i = 155\}$
$\mathcal{P}^r_{w,t}, r \in \{7,8,9\}$	$\{p_i : p_i \in \mathcal{P}^r_E, i = 234\}$

Table 4.4: Elements of watermarked programs in the training set for Experiment set A

The training and evaluation of $\mathfrak{detect}$ function are accomplished in the SPSS. The SPSS syntax used for $run_i, (1 \leq i \leq 9)$ is same and is presented in Appendix A.2.6.

The $\mathcal{G}^r$ used in $\mathcal{X}_A$ is:

$$\mathcal{G}^r \ = \ \mathcal{A}, 1 \leq r \leq 9$$

4.9 Design for Experiment Set B

Unlike the two-factor design of experiment set $\mathcal{X}_A$, $\mathcal{X}_B$ is a simple, one factor experiment. The primary factor used in $\mathcal{X}_B$ is the embedding combination $m_i, i \in \{1,2\}$ It consists of two $runs$: run_{13}, run_{14}

the $\mathfrak{transform}$ function (for $13 \leq r \leq 14$).:

$\mathcal{P}^r_{v,t}$	Elements
$\mathcal{P}^r_{w,v}, r \in \{1,4,7\}$	$\{p_i : p_i \in \mathcal{P}^r_E, i \ \in \{5,30,78,123,141,205\}\}$
$\mathcal{P}^r_{w,v}, r \in \{2,5,8\}$	$\{p_i : p_i \in \mathcal{P}^r_E, i \ \in \{15,16,18,69,87,197\}\}$
$\mathcal{P}^r_{w,v}, r \in \{3,6,9\}$	$\{p_i : p_i \in \mathcal{P}^r_E, i \ \in \{1,50,54,60,77,92\}\}$

Table 4.5: Elements of watermarked programs in the test set for Experiment set A

k	run
1	run_{10}
2	run_{11}
3	run_{12}

Table 4.6: experiment design for Experiment Set C

$$\mathfrak{transform}(p_{0,j}) = \begin{cases} p_j = \mathfrak{embed}(p_{0,j}, w_j, key_j, m_1) & if\ j \in \{1, 5, 15, 16, 18, 30, 50, 54, 60, \\ & 69, 73, 77, 78, 87, 92, 123, 141, 155, \\ & 197, 207, 234\}\ and\ r = 13 \\ p_j = \mathfrak{embed}(p_{0,j}, w_j, key_j, m_2) & if\ j \in \{1, 5, 15, 16, 18, 30, 50, 54, 60, \\ & 69, 73, 77, 78, 87, 92, 123, 141, 155, \\ & 197, 207, 234\}\ and\ r = 14 \end{cases}$$

4.10 Design for Experiment Set C

As with experiment set $\mathcal{X}_B$, $\mathcal{X}_C$ is also a simple, one-factor experiment. The primary factor used in $\mathcal{X}_C$ is the length of $k-grams$ used in $\mathcal{G}^r$. Table 4.6 shows the experiment design of $\mathcal{X}_C$.

For all runs in $\mathcal{X}_C$, $\mathcal{P}^r_{w,t}$, $\mathcal{P}^r_{w,v}$, $\mathcal{P}^r_{u,t}$, $\mathcal{P}^r_{u,v}$:

$$\mathcal{P}^r_{u,v} = \{p_i : p_i \in \mathcal{P}^r_E, i \in \{11, 14, 25, 35, 38, 49, 85, 93, 112, 133, 143, 148, 156, 163, 164, 174, 177, 189, 191, 196\}\}$$

$$\mathcal{P}^r_{u,t} = \{p_i : p_i \in \mathcal{P}^r_E, i \notin \{11, 14, 25, 35, 38, 49, 85, 93, 112, 133, 143, 148, 156, 163, 164, 174, 177, 189, 191, 196, 1, 5, 15, 16, 18, 30, 50, 54, 60, 69, 73, 77, 78, 87, 92, 123, 141, 155, 197, 207, 234\} r \in \{10, 11, 12\}\}$$

$$\mathcal{P}^r_{w,v} = \{p_i : p_i \in \mathcal{P}^r_E, i \in \{5, 15, 16, 18, 30, 50, 54, 60, 69, 73, 77, 78, 87, 92, 123, 141, 155, 197, 207, 234\} \; r \in \{10, 11, 12\}\}\}$$

$$\mathcal{P}^r_{w,t} = \{p_i : p_i \in \mathcal{P}^r_E, i = 1\} \; r \in \{10, 11, 12\}\}$$

As in $\mathcal{X}_A$, the *training* and *evaluation* of $\mathfrak{detect}$ function are accomplished in the SPSS. The SPSS syntax used for $run_i, (10 \leq i \leq 12)$ is presented in Appendix A.2.6. The $\mathcal{G}^{10}$ is listed between "/VARIABLES=" and "/ANALYSIS ALL" of SPSS syntax for 10-th *run* in Appendix A.2.6. The $\mathcal{G}^{11}$ is listed between "/VARIABLES=" and "/ANALYSIS ALL" of SPSS syntax for 11-th *run* in Appendix A.2.6. The $\mathcal{G}^{12}$ is listed between "/VARIABLES=" and "/ANALYSIS ALL" of SPSS syntax for 12-th *run* in Appendix A.2.6.

4.11 Violations of the Assumptions of Fisher's Discriminant Function

4.11.1 Violation of the Assumption of Multivariate Normality

Fisher's Discriminant Function assumes that variables are multivariate normality [28]. However, in practice, this assumption can often be violated [14]. According to [14, 28], this violation will not cause a big problem if the distribution of variables is not far away from the multivariate normality. Otherwise the results will not be accurate.

4.11.2 Violation of the Assumption of Equality of Variance-Covariance Matrices

Fisher's Discriminant Function also assumes that the variance-covariance matrices are equal [14, 28]. The violation of this assumption will cause the classification result not optimal [28]. However, according to [28], the classification result will still be good even this assumption is violated.

4.11.3 Violation of the Common Rule of Sample Size

In [14], a common rule about sample size is that "the number of cases in the smallest group should be five times the number of predictors [here predictor means feature]". However, we have only one sample in the watermarked program group in $\mathcal{P}_E$. So our feature number should be 0.2. This is impossible. Thus we violate this common rule and will make our results inaccurate.

5

Experiment Results and Analysis

5.1 Introduction

In this chapter, the experiment results are presented and analyzed. The validation of our experiment results are discussed in Section 5.4. Finally, we report some bugs of Sandmark in the CT watermark *extracting* process in Section 5.2.4.

5.2 Experiment Results

Experiment set $\mathcal{X}_A$ is used to verify the **Hypothesis One** which is stated in Chapter 4. Experiment set $\mathcal{X}_B$ is used to verify the **Hypothesis Two** and Experiment set $\mathcal{X}_C$ is used to verify the **Hypothesis Three**.

5.2.1 Results of Experiment set A

Training Results of Experiment set A

In this experiment, the attacker uses SPSS to build his/her $\mathfrak{detect}$ function. We present the Fisher's linear discriminant functions provided by SPSS as the attacker's $\mathfrak{detect}$ function. According to *SPSS 12.0 statistical procedures companion* p314 [28], "[the classification results by using the Fisher's linear discriminant functions] are identical to what you get using the discriminant function scores.". Since by using the discriminant function score, we can read the classification result of each case directly from the "predicted group" column of the "casewise result" Table of SPSS output, our experiment results are actually achieved by SPSS's discriminant function scores.

An example of how to classify the cases by using the Fisher's linear discriminant functions are given in Section 5.5. Here we give only the coefficients of Fisher's linear discriminant functions. Table 5.1, 5.2 and 5.3 present Fisher's linear discriminant function coefficients for (run_1, run_2, run_3), (run_4, run_5, run_6),(run_7, run_8, run_9) respectively, where function 1 is the function for unwatermarked program class and function 2 is the function for watermarked program class. We present Fisher's linear discriminant functions as attacker's $\mathfrak{detect}^{\mathfrak{r}}$ functions for run_1 to run_9.

Table 5.1: Fisher's Linear Discriminant Function Coefficients for run_1, run_2, run_3

$1-gram$	**Function1**	**Function2**	$1-gram$	**Function1**	**Function2**
dcmpg	476922.2	472120.5	dsub	-135886.2	-125462
l2i	16548	14620.7	fastore	227475	216594.6
aastore	10488.5	11509.1	ldc	6121.2	6433.9
dcmpl	2462.7	3639.8	iaload	-5885	-5186
dload_0	-65024.7	-72776.2	iload_0	22702.3	24403.6
dload_1	93611.5	92531.2	i2l	17034.7	26645.8

Continued on next page

Table 5.1 – continued from previous page

$1-gram$	**Function1**	**Function2**	$1-gram$	**Function1**	**Function2**
fload_0	-192193.2	-134889.8	daload	-39713.1	-38936.7
dload_2	68557.5	87797.4	iload_1	-80.4	-213.8
fload_1	700803.1	637704.2	iload_2	-4934.5	-2073.4
iand	35281	32629.9	iload_3	25430.2	26831.9
dload_3	-48879.3	-42993.4	lsub	150688.5	132959.9
fload_2	-841932.7	-767289.3	dreturn	-183709.1	-196458.9
fload_3	1345628.2	1270922.3	dadd	-24681.2	-37104.3
lreturn	240247	199354.5	i2s	-105045.1	-94941.8
athrow	28253.5	25618.3	bastore	315.3	577.3
dmul	139813.9	145706.1	lxor	974939.3	786790.4
ifle	9039	5161.4	imul	40271.2	40512.9
lastore	59644.9	31241.8	dastore	-4181	-7939.3
aaload	16046.2	15268.5	new	-22617.7	-21840.3
anewarray	-55662.6	-55806	ifge	108769.2	115556.7
if_icmple	-20871.8	-17842.1	ladd	-387013.3	-374373.3
irem	222357.5	217004.9	invokeinterface	20184.7	20356.7
pop	12019.8	11451.8	if_icmpge	-22637.6	-23760.7
checkcast	26302.3	25978.4	iushr	124828.5	118715.7
fsub	191976.5	184478	sipush	2367	2198
lmul	129142.4	101798.9	monitorenter	44515.1	49381.9
putfield	15801.6	15639.2	ifeq	27483	26771.2
ifne	-21458.6	-19873.4	dconst_0	643.1	2537.5
ifnonnull	365.5	1208.7	dconst_1	70273.6	67280.4
saload	-42626	-62464.3	if_icmpeq	7400.4	6618.5
invokespecial	11763.5	11829.2	tableswitch	-9033.2	6148.2
if_icmpne	15257.6	16735.3	frem	142712.6	935103.5
fcmpg	9954.3	40359.1	dup_x1	-59551.1	-56331.6

Continued on next page

Table 5.1 – continued from previous page

$1-gram$	**Function1**	**Function2**	$1-gram$	**Function1**	**Function2**
ineg	-98850	-108921.8	newarray	12366.1	12123.1
fstore	200509.1	202692	dup_x2	125084.1	132111.1
fadd	-545213.4	-537826.4	caload	52584.8	51607.5
fcmpl	-1021941.1	-999832.2	fconst_0	70535.3	69818.6
d2f	-591314.6	-568719.1	istore	32990.3	29878.8
idiv	44566.3	46227.7	fconst_1	-350456.2	-375150.4
astore	13783.6	14201.1	ifgt	-34233.8	-48032.7
instanceof	20769.2	19358.8	fconst_2	170927.9	256264.9
iflt	21879.3	9887.6	ior	-94967.2	-78994.2
bipush	1584.5	1122.2	putstatic	-2445.8	-648.8
if_icmplt	54304.1	53024.3	if_icmpgt	-53659.4	-53725
d2i	-189346.5	-190576.5	dstore	62482.1	68946.3
invokestatic	19405.7	18392	iconst_0	346.5	-78.8
invokevirtual	12984.7	12494.7	lshl	-36751.9	-161477
lor	338583.7	438751.7	iconst_1	19775.3	18489.6
dup2_x1	-190002.7	-255484.3	iconst_2	2619.9	984.9
lookupswitch	45703.6	50568.5	getfield	1938.6	1541.1
d2l	2358637.9	2215489.6	fneg	882378.5	673188.1
dup2_x2	294447	392300.8	iconst_3	15512.1	16182.2
if_acmpeq	52537.9	51169	lconst_0	93115.9	90934.7
i2b	138389.3	121263.9	iconst_4	-43640.9	-36837.8
freturn	72176.7	77327.8	dup	22675.6	22489.1
i2c	-3479.3	-4554.5	lconst_1	155313.5	207031.9
return	2428.7	1438.2	f2l	-1014953	-1197502.5
i2d	9394	9948.8	aload_0	8400.6	8714.8
ret	-210147.6	-200269.8	ireturn	3582.4	1895.7
i2f	-35559.8	-39951.6	iinc	-24575.8	-20772.7

Continued on next page

Table 5.1 – continued from previous page

$1-gram$	**Function1**	**Function2**	$1-gram$	**Function1**	**Function2**
sastore	-73249	-51515.6	castore	-5759.2	-4877.7
dload	-21514.1	-26032.6	(Constant)	-3330.8	-3282.8
lload	-112180.4548	-99961.26121			

Table 5.2: Fisher's Linear Discriminant Function Coefficients for run_4, run_5, run_6

$1-gram$	**Function1**	**Function2**	$1-gram$	**Function1**	**Function2**
dcmpg	476922.2	534027.2	dsub	-135886.2	-137199.5
l2i	16548	4966.6	fastore	227475	237511.8
aastore	10488.5	11526.2	ldc	6121.2	5573.2
dcmpl	2462.7	590.3	iaload	-5885	-4494.6
dload_0	-65024.7	-68056.3	iload_0	22702.3	23021.5
dload_1	93611.5	87806.8	i2l	17034.7	11510.7
fload_0	-192193.2	-216809.1	daload	-39713.1	-38031.8
dload_2	68557.5	55665.9	iload_1	-80.4	640.5
fload_1	700803.1	718794	iload_2	-4934.5	-2418.5
iand	35281	33683	iload_3	25430.2	27445.7
dload_3	-48879.3	-49352.4	lsub	150688.5	172971.1
fload_2	-841932.7	-855240.6	dreturn	-183709.1	-170914.9
fload_3	1345628.2	1394908.2	dadd	-24681.2	-12936.5
lreturn	240247	280732.1	i2s	-105045.1	-116120.8
athrow	28253.5	26714.7	bastore	315.3	1327.3
dmul	139813.9	141792.1	lxor	974939.3	606330.6
ifle	9039	8319.9	imul	40271.2	42581
lastore	59644.9	136080.5	dastore	-4181	-3042

Continued on next page

Table 5.2 – continued from previous page

$1-gram$	**Function1**	**Function2**	$1-gram$	**Function1**	**Function2**
aaload	16046.2	15491.4	new	-22617.7	-20692.8
anewarray	-55662.6	-54542.5	ifge	108769.2	107385
if_icmple	-20871.8	-12813.7	ladd	-387013.3	-436632.4
irem	222357.5	242953.6	invokeinterface	20184.7	18636
pop	12019.8	12579	if_icmpge	-22637.6	-16237.3
checkcast	26302.3	26105.6	iushr	124828.5	132889.4
fsub	191976.5	194177.8	sipush	2367	657.2
lmul	129142.4	150757	monitorenter	44515.1	42503.4
putfield	15801.6	16961.5	ifeq	27483	26797.4
ifne	-21458.6	-22573	dconst_0	643.1	250.6
ifnonnull	365.5	-932.2	dconst_1	70273.6	62733.6
saload	-42626	-11191.3	if_icmpeq	7400.4	9468.1
invokespecial	11763.5	10705.1	tableswitch	-9033.2	-23406.8
if_icmpne	15257.6	9753.6	frem	142712.6	548499.5
fcmpg	9954.3	66108.5	dup_x1	-59551.1	-61168.1
ineg	-98850	-97338.5	newarray	12366.1	14654.3
fstore	200509.1	184600.3	dup_x2	125084.1	83772.4
fadd	-545213.4	-539713	caload	52584.8	57185
fcmpl	-1021941.1	-1005639	fconst_0	70535.3	67184.5
d2f	-591314.6	-568592.6	istore	32990.3	34157.8
idiv	44566.3	37635.9	fconst_1	-350456.2	-348727.4
astore	13783.6	15524.3	ifgt	-34233.8	-77254.2
instanceof	20769.2	20982.8	fconst_2	170927.9	101995.3
iflt	21879.3	17253	ior	-94967.2	-109865.6
bipush	1584.5	882.1	putstatic	-2445.8	-3207.9
if_icmplt	54304.1	58915.1	if_icmpgt	-53659.4	-53991.8
d2i	-189346.5	-202074.7	dstore	62482.1	48810

Continued on next page

Table 5.2 – continued from previous page

$1-gram$	**Function1**	**Function2**	$1-gram$	**Function1**	**Function2**
invokestatic	19405.7	20209.7	iconst_0	346.5	-867.8
invokevirtual	12984.7	12497.7	lshl	-36751.9	-7836.8
lor	338583.7	361662.4	iconst_1	19775.3	19706.3
dup2_x1	-190002.7	-269845.4	iconst_2	2619.9	-2244.6
lookupswitch	45703.6	34488	getfield	1938.6	1903.8
d2l	2358637.9	2597313.2	fneg	882378.5	610601.3
dup2_x2	294447	247840.2	iconst_3	15512.1	10820.4
if_acmpeq	52537.9	49646.8	lconst_0	93115.9	104197.8
i2b	138389.3	150428.9	iconst_4	-43640.9	-37313.4
freturn	72176.7	-13795.8	dup	22675.6	23137
i2c	-3479.3	-8626.4	lconst_1	155313.5	152153
return	2428.7	1029.1	f2l	-1014953	-1238374.3
i2d	9394	9603	aload_0	8400.6	8192.6
ret	-210147.6	-223124.9	ireturn	3582.4	3400.3
i2f	-35559.8	-39197	iinc	-24575.8	-33504
sastore	-73249	-71275.7	castore	-5759.2	-5229.5
dload	-21514.1	-20779	(Constant)	-3330.8	-3341.8
lload	-112180.5	-127434.5			

Table 5.3: Fisher's Linear Discriminant Function Coefficients for run_7, run_8, run_9

$1-gram$	**Function1**	**Function2**	$1-gram$	**Function1**	**Function2**
dcmpg	476922.2	644686.9	dsub	-135886.2	-128950.4
l2i	16548	14276.9	fastore	227475	192810.8
aastore	10488.5	12652	ldc	6121.2	5694.6

Continued on next page

Table 5.3 – continued from previous page

$1-gram$	**Function1**	**Function2**	$1-gram$	**Function1**	**Function2**
dcmpl	2462.7	-14440.6	iaload	-5885	1267.7
dload_0	-65024.7	-49586.7	iload_0	22702.3	25900.8
dload_1	93611.5	78746.8	i2l	17034.7	-12646.4
fload_0	-192193.2	-53678.9	daload	-39713.1	-26834.4
dload_2	68557.5	100059	iload_1	-80.4	2078.6
fload_1	700803.1	628635.2	iload_2	-4934.5	4254.5
iand	35281	14744	iload_3	25430.2	26826
dload_3	-48879.3	-52303.1	lsub	150688.5	119179.3
fload_2	-841932.7	-798799.9	dreturn	-183709.1	-174666.9
fload_3	1345628.2	1333213.3	dadd	-24681.2	-11892.3
lreturn	240247	242202.6	i2s	-105045.1	-111326.5
athrow	28253.5	18241.8	bastore	315.3	2280.9
dmul	139813.9	143690.7	lxor	974939.3	180695.1
ifle	9039	4366.5	imul	40271.2	42536.4
lastore	59644.9	136577.7	dastore	-4181	-4263.9
aaload	16046.2	12468.5	new	-22617.7	-13903.8
anewarray	-55662.6	-50538.8	ifge	108769.2	123483.5
if_icmple	-20871.8	-3302.1	ladd	-387013.3	-302530.9
irem	222357.5	221135.7	invokeinterface	20184.7	16522.3
pop	12019.8	11427	if_icmpge	-22637.6	-17487.3
checkcast	26302.3	21160.5	iushr	124828.5	137726.2
fsub	191976.5	234358.7	sipush	2367	-860.4
lmul	129142.4	78380.7	monitorenter	44515.1	48259.7
putfield	15801.6	17867.4	ifeq	27483	21919.5
ifne	-21458.6	-18461.6	dconst_0	643.1	3929.7
ifnonnull	365.5	-3043.7	dconst_1	70273.6	47882.6
saload	-42626	30160	if_icmpeq	7400.4	15920.3

Continued on next page

Table 5.3 – continued from previous page

$1-gram$	**Function1**	**Function2**	$1-gram$	**Function1**	**Function2**
invokespecial	11763.5	6784.2	tableswitch	-9033.2	1912.1
if_icmpne	15257.6	5791.6	frem	142712.6	2816745.4
fcmpg	9954.3	216015	dup_x1	-59551.1	-35846.1
ineg	-98850	-72522.4	newarray	12366.1	17647.6
fstore	200509.1	169539.7	dup_x2	125084.1	30538.9
fadd	-545213.4	-473625.5	caload	52584.8	60049.3
fcmpl	-1021941.1	-1018621.1	fconst_0	70535.3	61890.5
d2f	-591314.6	-566005.6	istore	32990.3	29054.4
idiv	44566.3	36778	fconst_1	-350456.2	-358347.6
astore	13783.6	19034	ifgt	-34233.8	-100860.6
instanceof	20769.2	23045.8	fconst_2	170927.9	212747
iflt	21879.3	-10649.7	ior	-94967.2	-105647.2
bipush	1584.5	-972.9	putstatic	-2445.8	3563.9
if_icmplt	54304.1	67782.7	if_icmpgt	-53659.4	-54182.4
d2i	-189346.5	-204650	dstore	62482.1	34601.2
invokestatic	19405.7	18572.2	iconst_0	346.5	-3666.5
invokevirtual	12984.7	10356.8	lshl	-36751.9	-220305.1
lor	338583.7	591269.4	iconst_1	19775.3	15180.9
dup2_x1	-190002.7	-490280.5	iconst_2	2619.9	-2210.6
lookupswitch	45703.6	45841.6	getfield	1938.6	1568.4
d2l	2358637.9	2431321.2	fneg	882378.5	-368822.4
dup2_x2	294447	1130726.9	iconst_3	15512.1	1088
if_acmpeq	52537.9	38329.1	lconst_0	93115.9	120557.2
i2b	138389.3	141136.5	iconst_4	-43640.9	-18709.6
freturn	72176.7	-6055.3	dup	22675.6	23060.4
i2c	-3479.3	-13408	lconst_1	155313.5	131140.7
return	2428.7	-54.4	f2l	-1014953	-1792131

Continued on next page

	QVL_1	QVL_2	QVL_3
QTL_1	50	50	16.7
QTL_2	83.3	66.7	0
QTL_3	100	66.7	0

Table 5.4: FN of Detecting CT watermark of Experiment Set A
(See Table 4.2 for the meaning of QTL_i and QVL_j, $i, j \in \{1, 2, 3\}$.)

	QVL_1	QVL_2	QVL_3
QTL_1	35	35	35
QTL_2	0	0	0
QTL_3	0	0	0

Table 5.5: FP of Detecting CT watermark of Experiment Set A
(See Table 4.2 for the meaning of QTL_i and QVL_j, $i, j \in \{1, 2, 3\}$.)

Table 5.3 – continued from previous page

$1-gram$	**Function1**	**Function2**	$1-gram$	**Function1**	**Function2**
i2d	9394	21903.2	aload_0	8400.6	7959.1
ret	-210147.6	-196373.8	ireturn	3582.4	2316.3
i2f	-35559.8	-68318.3	iinc	-24575.8	-32774.7
sastore	-73249	-70324.8	castore	-5759.2	-3893
dload	-21514.1	-39994.9	(Constant)	-3330.8	-3687.8
lload	-112180.5	-110057.3			

Evaluation Results of Experiment set A

Table 5.4 and Table 5.5 present the false negative rate and false positive rate of attacker's **detect** function in each *run* of experiment set A. They are our main experiment outputs. The QTL_i and QVL_j in the two tables are the levels of watermark size ratios of $\mathcal{P}^r_{w,t}$ and $\mathcal{P}^r_{w,v}$ respectively, where $i, j \in \{1, 2, 3\}$. The values of QTL_i and QVL_j are presented in Table 4.2 in Chapter 4.

Tables 5.4 and Table 5.5 show some interesting results:

First, we can judge the success of the detection attack based on the judgement rules discussed in Chapter 2. Table 5.6 shows the results of judgement.

		QTL_1 ($\forall q \in \mathcal{Q}^r_{w,t}$, $0 < q \leq 0.2$)	QTL_2 ($\forall q \in \mathcal{Q}^r_{w,t}$, $0.2 < q \leq 2.0$)	QTL_3 ($\forall q \in \mathcal{Q}^r_{w,t}$, $2.0 < q$)
QVL_1	($\forall q \in \mathcal{Q}^r_{w,v}$, $0 < q \leq 0.2$)	total failure	total failure	weak success
QVL_2	($\forall q \in \mathcal{Q}^r_{w,v}$, $0.2 < q \leq 2.0$)	strong success	strong success	total success
QVL_3	($\forall q \in \mathcal{Q}^r_{w,v}$, $2.0 < q$)	strong success	strong success	total success

Table 5.6: Judging the Success of Attackers

According to Table 5.6, the combinations of (QTL_2, QVL_3) and (QTL_3, QVL_3) correspond to a high watermark size ratio level in both of $\mathcal{P}_{w,t}$ and $\mathcal{P}_{w,v}$. The combinations of (QTL_1, QVL_1) and (QTL_1, QVL_2) correspond to a low watermark size ratio level in both of $\mathcal{P}_{w,t}$ and $\mathcal{P}_{w,v}$. The first combinations cause total success of the attacker and the second combinations cause total failure of the attacker. Or we can say the first combinations have a low error rate (FN and FP) of the attacker's $\mathfrak{detect}$ function while the second have a high error rate. Thus we can draw two conclusions: (1). high watermark size ratio in both of $\mathcal{P}_{w,t}$ and $\mathcal{P}_{w,v}$ will have low error rate; (2). low watermark size ratio in both of $\mathcal{P}_{w,t}$ and $\mathcal{P}_{w,v}$ will have a high error rate. Thereby we proving that **Hypothesis One** stated in Chapter 4 is true.

5.2.2 Results of Experiment set B

Table 5.7 shows the pair of watermark size ratios by using m_1 and by using m_2. Table 4.9 has the procedure for preparing watermarked programs for experiment set B by using the $\mathfrak{transform}$ function (used for $13 \leq r \leq 14$).

The mean value of using m_1 in embedding is 1.582, and the mean value of using m_2 in embedding is 0.135. We use the *paired-samples t-test* to check whether there is a significant difference between using m_1 and m_2. *Paired-samples t-test* is a test to measure whether the mean of two sets of data indicates significant difference. The test result is $Sig.(2-tails) = 0.008$. According to Julie [29], when Sig. (2-tails) is less than 0.05,

Name	m_1	m_2
TTT	2.794	0.233
web_ActiveRegionExplorer	0.047	0.004
web_aspectj-1.1.0	0.783	0.071
web_AutoSim	0.269	0.019
web_BBI	0.222	0.017
web_ChronicleLite-bin-v1.2	0.142	0.012
web_DigestCalc	2.247	0.196
web_dss	10.022	0.872
web_ff	3.078	0.250
web_fuzzyide	0.267	0.023
web_grades	0.089	0.007
web_hqt	3.063	0.252
web_HTMLEditorPro	0.054	0.005
web_jar-util	0.646	0.056
web_javapopt	2.058	0.165
web_jpp	0.094	0.008
web_Logisim	0.152	0.013
web_ModEdit	0.429	0.039
web_run	0.809	0.068
web_SpidersRUs	0.073	0.006
web_YMStrings	5.894	0.510

(See Table 4.1 for the meaning of m_1 and m_2)

Table 5.7: Watermark size ratios for two embedding parameter combinations

the two means are significantly different. So we can conclude that the mean values of *watermark size ratio*s by using m_1 and by using m_2 display significant difference. The mean value of *watermark size ratio*s by using m_2 (*using numeric watermark*) is much lower than by using m_1 (*using cycle graph*). Thus we prove that our **Hypothesis Two** stated in Section 4.6 is true.

5.2.3 Results of Experiment set C

Training Results of Experiment set C

Tables 5.8, 5.9 and 5.10 show the Fisher's Linear Discriminant Function Coefficients for run_{10}, run_{11} and run_{12} respectively. We present them as the attacker's trained $\mathfrak{detect}^r$ functions for run_{10}, run_{11} and run_{12}.

Table 5.8: Fisher's Linear Discriminant Function Coefficients for run_{10}

$1-gram$	**Function1**	**Function2**
putfield	5.6	1110
astore	126	1754
invokevirtual	17.6	-236.3
new	267.9	1067.9
istore	193.8	-1685.7
isub	77.5	949.5
aload_0	76.9	-218.9
aload_2	88.9	-310.1
iadd	131.5	567.6
istore_1	549.3	3156.9
istore_2	754.6	3946.1
iload	44.1	537.3
(Constant)	-15.8	-213.3

Table 5.9: Fisher's Linear Discriminant Function Coefficients for run_{11}

(Note that A.B is a 2-gram where A is the first opcode and B is the second opcode in the 2-gram)

$2-gram$	**Function1**	**Function2**
iconst_3.invokespecial	539.3	356642.1
istore_1.iconst_0	2655.2	557397.1

Continued on next page

Table 5.9 – continued from previous page

$2-gram$	**Function1**	**Function2**
goto.ldc	476.5	76744.5
iadd.iload_2	32.7	-155521
iadd.iload_3	833.5	-459167.4
istore_1.getstatic	-766.2	-1164691
aload.bipush	179.5	116543
if_icmple.iconst_m1	-931	1832448.4
iconst_2.if_icmpne	-51.7	-179596.5
iconst_1.goto	225.2	168948.4
iload_1.invokestatic	223.5	-376313.9
getfield.astore	507.7	155456.8
getfield.getstatic	-76.1	-142756.4
iconst_0.invokestatic	116.9	-78200.4
goto.iconst_2	3338	879442.4
ior.istore_1	287.5	333125.1
iload.iload_3	-228.9	-291383.1
ifne.iconst_1	-1423.5	-767040.1
putfield.new	1215.3	363627.1
ifeq.getstatic	-112.5	-90528.3
iand.aload_0	-996.3	697134.2
aaload.ldc	29.4	95531.7
aastore.iinc	-353.6	-269206.6
invokespecial.ior	53685.6	119603675.6
getfield.iload	221.8	42991.1
aload.putfield	313.8	198109.7
putstatic.goto	39.7	55448.2
astore.new	-76	-37434.6
aload_1.astore_2	1214.9	-408697.1

Continued on next page

Table 5.9 – continued from previous page

$2-gram$	**Function1**	**Function2**
ldc.aload_0	306.2	71835.8
iconst_0.istore	127.3	20986.5
dup.invokespecial	208	11070.7
ldc2_w.lcmp	-437.3	-641268.2
(Constant)	-4.6	-145287.1

Table 5.10: Fisher's Linear Discriminant Function Coefficients for run_{12}

(Note that A.B.C is a 3-gram where A is the first opcode, B is the second opcode and C is the third opcode in the 3-gram)

$3-gram$	**Function1**	**Function2**
if_acmpne.iconst_1.istore_1	8374.6	-125498914.1
aload_0.iconst_0.iconst_4	-10793.5	-246783472.7
invokevirtual.aload_0.ldc	31.5	-147381.6
getfield.invokevirtual.ldc	108.5	146225.1
getstatic.invokevirtual.invokespecial	-31.2	-5132792.4
ldc.if_acmpne.iconst_1	-12198.6	-135617638.5
iconst_2.iconst_4.bipush	-69	-60351781.5
bipush.invokevirtual.checkcast	262.5	-3656266.3
iconst_2.iadd.invokespecial	3333.9	-15712319.8
astore_1.aload_1.astore_2	-132.4	6240592.2
iconst_1.istore_1.iinc	2629.4	-18275143.5
invokestatic.getstatic.iload_2	3006.9	64282763.2

Continued on next page

k	1	2	3
FN (%)	55	100	25
FP (%)	0	0	0

Table 5.11: The Effect of Length of $k-gram$ (k) on FN and FP of attacker's $\mathfrak{detect}$ function

Table 5.10 – continued from previous page

$3-gram$	**Function1**	**Function2**
getfield.getstatic.invokevirtual	101.5	233769
ldc.aload_0.getfield	120.8	956688.5
aload.putfield.new	13892.9	249736278.4
getfield.ifne.ldc	-2290.3	-28561715
iconst_2.putfield.aload_0	729.9	1035831.7
bipush.invokespecial.invokevirtual	96	164465
iadd.iload_3.iconst_2	6411.4	89065532.6
istore_2.iload_2.iflt	921.2	-2860894.6
aload_0.ldc.aload_0	-441.8	-3340396.5
aaload.iload_1.invokestatic	1816.4	168707706.8
dup.invokespecial.ldc	61	50312.8
nop.goto.nop	-645.8	-20926818.6
invokevirtual.ifeq.getstatic	78.5	-269162.9
(Constant)	-1.8	-9250570.8

Evaluation Results of Experiment set C

Table 5.11 shows how the selection of k for $\mathcal{G}^r$ will affect FN^r and FP^r of the attacker's $\mathfrak{detect}$ function. For the same $\mathcal{P}_t^r$ and $\mathcal{P}_v^r$, FN^r changes when k changes. Thus we prove our **Hypothesis Three** stated in Section 4.6. But the FN^r is not always lower when k is higher. e.g.when $k = 2$, $FN = 100\%$. However, when $k = 1$, $FN = 55\%$.

Watermark	Recognition behaviour
123	recognized
metalworks	can not be recognized
metal123	can not be recognized
qsc123	recognized
m123	recognized
qsc12345	can not be recognized
qsc1234	recognized only in command line

Table 5.12: CT Watermark Recognition Results of a Certain Program (called “metalworks”)

5.2.4 Bug Report

In our experiment environment, we find bugs in the Sandmark in CT watermark extracting process. The experiment environment is presented in Appendix A.1. The bug is that Sandmark sometimes can not correctly recognize the watermark from CT watermarked programs. A case illustrating the bug is as follows.: when embedding CT watermarks into a program called “metalworks”, some embedded watermarks can be successfully recognized while other watermarks can not be recognized. An interesting fact is that watermark “qsc1234” can be recognized in the command line interface of Sandmark. But the graph interface of Sandmark can not report the recognized watermark. The bug information is given in Table 5.12. The *key* (user's input sequence) is : *file* $\Rightarrow$ *new* $\Rightarrow$ *close* $\Rightarrow$ *quit*

5.3 Results Analysis

5.3.1 Analyzing Results for Experiment Set A

The experiment results in Section 5.2.1 show that the success of detection watermark attack relies on the watermark size ratios of CT watermarked programs. High watermark size ratio in both of $\mathcal{P}_{w,t}$ and $\mathcal{P}_{w,v}$ will tend to give low FN and FP for attacker's $\mathfrak{detect}$ function. So we should control the watermark size ratio of watermarked programs lower than a certain criterion to defend against *AUDW*.

Our experiment results show that when every watermarked program in $\mathcal{P}_{w,v}$ and $\mathcal{P}_{w,t}$

has a watermark size ratio higher than 2.0 (the combination of (QVL_3, QTL_3)), the attackers $\mathfrak{detect}$ function will have 0% FN and 0% FP. It will be the total success of the attacker. So at least the criterion of watermark size ratio should be less than 2.0. More experiments are needed to determine the exact criterion of the watermark size ratio which is beyond the scope of our thesis.

To amend above problem, our suggestion is: from the equation to calculate the watermark size ratio q of a watermarked program p_w, we can find two ways to restrict q under a criterion β. $\beta \in \mathcal{Q}$, $0 < \beta$.

The ratio q can be calculated as follows.

$$\mathfrak{ratio}(p_w) \;=\; \frac{|p_w| - |p_u|}{|p_u|}$$

where $p_w \;= embed(p_u, w, key, m)$; w is the watermark to be embedded; key is the input sequence used to embed watermark and m is the combinations of parameters used in watermark embedding. $|p_w|$ and $|p_u|$ are the size of p_w and p_u.

So to make sure $q \;<\; \beta$ means

$$\begin{aligned} q \;<\; \beta \Rightarrow \frac{|embed(p_u, w, key, m)| - |p_u|}{|p_u|} &<\; \beta \\ \Rightarrow |embed(p_u, w, key, m)| - |p_u| &<\; \beta|p_u| \\ \Rightarrow |embed(p_u, w, key, m)| &<\; (\beta+1)|p_u| \\ \Rightarrow \frac{|embed(p_u, w, key, m)|}{(\beta+1)} &<\; |p_u| \end{aligned} \tag{5.1}$$

We can either fix $|p_u|$ and reduce $\frac{|embed(p_u,w,key,m)|}{(\beta+1)}$ or make sure $|p_u| > \frac{|embed(p_u,w,key,m)|}{(\beta+1)}$.

To reduce $\frac{|embed(p_u,w,key,m)|}{(\beta+1)}$, we should make our watermark embedding more efficient. According to Collberg and Townsend [7], in Sandmark, when we embed a watermark using the "*numeric watermark*" option, the watermark embedded will be more efficient. We proved that by using the "*numeric watermark*" option in the embedding process then the watermark size ratio of a watermarked program is far lower than by using the "*cycle graph*" option. So using the "*numeric watermark*" option is one way to reduce watermark

size ratio. The drawback of using "*cycle graph*" option is that once we opt for *cycle graph*, we can only input pure digital numbers. We can not input characters such as "a", "b", "c", ... So it will reduce the number of watermarks we can use.

Another way is to make sure that $|p_u| > \frac{|embed(p_u,w,key,m)|}{(\beta+1)}$. If there is a program p_u, we can not find a way to make sure $|p_u| > \frac{|embed(p_u,w,key,m)|}{(\beta+1)}$, we can refuse to watermark this program by claiming such a kind of programs are not suitable for our watermark algorithm.

5.4 Discussion

As we discussed in Section 2.3.2, our experiment results only have limited meaning. Our experiment results are only valid for our sample set and our experiment method.

In addition, we design our experiments in an exploring process, the experiment parameters did not keep consistent. For example, in experiment set $\mathcal{X}_A$, we did not use watermarks consist of pure numeric numbers. However, when we tried to perform experiment set $\mathcal{X}_B$, we find only pure numeric numbers can be accepted as watermarks when we use *numeric watermark* option to embed watermarks. So the watermark set used in $\mathcal{X}_A$ and $\mathcal{X}_B$ are not consistent.

5.5 Example of Using the Detect Function

An example of using Fisher's linear discriminant function to classify a program is as follows:

Suppose we have a program p and we retrieved the pattern vector $\overrightarrow{p}^r$ by using $\mathcal{G}^r$. Suppose $\mathcal{G}^r = \{dcmpg, dsub, aastore\}$, $\overrightarrow{p}^r = \{0.1, 0.2, 0.7\}$, where the first element of $\overrightarrow{p}^r$ is the relative frequency of the occurrence of *dcmpg* in p, the second element of $\overrightarrow{p}^r$ is the relative frequency of the occurrence of *dsub* in p, and so on.

From Fisher's linear discriminant functions in Table 5.13, Function1 is for class1 (unwatermarked program class) and Function2 is for class2 (watermarked program class).

$1-gram$	Function1	Function2
dcmpg	100	10
dsub	20	20
aastore	50	40
(Constant)	-5	-2

Table 5.13: Example of Using Fisher's Linear Function to Classify Case

Then we can classify p as follows.

by using Function 1,

$$y_1 = 100 \times 0.1 + 20 \times 0.2 + 50 \times 0.7 - 5 = 44$$

By using Function 2,

$$y_2 = 10 \times 0.1 + 20 \times 0.2 + 40 \times 0.7 - 2 = 31$$

Since $y_2 > y_1$, we classify p to class2, the watermarked program class.

6

Conclusion and Future Work

6.1 The Challenge for the CT Watermark Designer and AUDW Attacker

For AUDW, the challenge for the attacker is that he/she should do his/her best to reduce the error rate of the $\mathfrak{detect}$ function. On the other hand, to defeat the assault from an attacker, the CT watermark designer should find ways to increase the error rate of the attacker's $\mathfrak{detect}$ function.

For attackers using the pattern classification methods, decisions made in pattern classification procedures will affect the error rate of attackers' detect function. So a good attacker will try to reduce the error rate of his/her $\mathfrak{detect}$ function by optimizing the

pattern classification activities such as *data collection*, *feature choice*, *model choice*, etc..

To simplify our experiment, we only simulate an attacker implementing Fisher's discriminant function by using the statistical analyzing tool called SPSS. We focus on the effects of two factors which will change the error rate of the attacker's $\mathfrak{detect}$ function: the *watermark size ratio*s of the watermarked programs collected by the attacker and the length of $k-grams$ used by the attacker to retrieve the pattern vectors. The results show that the error rate of attacker's $\mathfrak{detect}$ function is low when both of the watermark size ratio of the watermarked programs used to train the attacker's $\mathfrak{detect}$ function and to evaluate the attacker's $\mathfrak{detect}$ function are high.

We suggest two ways to solve above problem: using *numeric watermark* option in the CT watermark embedding procedure or restrict the size of the program to be watermarked. Our experiment show that the watermark size ratio of watermark programs will be dramatically reduced by using *numeric watermark* embedding option.

The length of $k-grams$ used by the attacker to retrieve the pattern vectors will also affect the error rate of attacker's $\mathfrak{detect}$ function. However, as discovered in our experiment, increasing the length of $k-grams$ does not always reduce the error rate of attacker's $\mathfrak{detect}$ function.

6.2 Future Work

Our experiments are only in the early stage of research into attacks on the CT watermark. We implemented all only one type of AUDW due to the limitations of this thesis. The accuracy of the attacker's $\mathfrak{detect}$ functions by using other popular classification algorithms is still unknown. More experiments should be done to verify whether those suggested obfuscation methods can really defeat our attack.

On the theoretic level, we need to carefully define the formal attack model analogous to what is suggested in media watermarking field by Mauro Barni et al [2]. WE did some early stage developments on this topic but it is far away from accomplishment.

Our experiments reveal the information hiding problem in software watermarking. In

media watermarking, a similar problem are researched by Cox [12] and [24]. However, their results can not be used in our AUDW. They did not consider the detection of watermark attacks in their research. So we should solve the problem ourself.

A

Appendix

A.1 Experiment Environment

A.1.1 Software System

System files:

Microsoft Window XP Professional Version 2002 Service Pack 2

Linux:Version: Linux 2.4.27-2-686-smp

Java SDK: Java TM 2 Platform Standard Edition 5.0 Development Kit (JDK 5.0)

Tool for obtaining pattern frequencies (My programs) The tools for obtaining pattern frequencies are some shell files and gawk files:.

Gawk Version: 3.1.4

Bash version: 3.1.0(1)-release

Tool for statistical analysis (SPSS)

SPSS Version: 14.0 for windows Release 14.0.0 (5 Sep 2005)

Tool for producing C-T fingerprinted programs Tools for annotating programs (which is from our group member Jasvir).

CTifier : Created On Thu Dec 8 13:24:27 2005

The Library files used by "CTifier" are:

BCEL: Version 5.1

Sandmark:Version 3.4.0

Tool for embedding C-T watermark:.

Sandmark:Version 3.4.0

Tools for obtain pattern vectors :.

Programs developed by myself: "OPTfilepretest","OPTdatacollection" and "OPTRuntransform".

A.1.2 Hardware

CPU: Intel(R) Pentium(R) 4 CPU 2.80GHz Memory: 1.00GB of RAM

A.2 Experiment Procedure

A.2.1 Java opcode set

Table A.1: Java Virtual Machine Opcode Set

Elements	Opcodes	Elements	Opcodes
op_1	dcmpg	op_{104}	fconst_1
op_2	l2i	op_{105}	ifgt
Continued on next page			

Table A.1 – continued from previous page

Elements	Opcodes	Elements	Opcodes
op_3	aastore	op_{106}	fconst_2
op_4	dcmpl	op_{107}	ior
op_5	dload_0	op_{108}	goto_w
op_6	dload_1	op_{109}	putstatic
op_7	fload_0	op_{110}	if_icmpgt
op_8	dload_2	op_{111}	dstore
op_9	fload_1	op_{112}	iconst_0
op_{10}	iand	op_{113}	lshl
op_{11}	dload_3	op_{114}	iconst_1
op_{12}	fload_2	op_{115}	iconst_2
op_{13}	fload_3	op_{116}	getfield
op_{14}	lreturn	op_{117}	fneg
op_{15}	impdep1	op_{118}	iconst_3
op_{16}	athrow	op_{119}	fload
op_{17}	impdep2	op_{120}	lconst_0
op_{18}	dmul	op_{121}	iconst_4
op_{19}	ifle	op_{122}	f2d
op_{20}	lastore	op_{123}	dup
op_{21}	aaload	op_{124}	lconst_1
op_{22}	anewarray	op_{125}	iconst_5
op_{23}	if_icmple	op_{126}	getstatic
op_{24}	irem	op_{127}	fdiv
op_{25}	pop	op_{128}	lshr
op_{26}	checkcast	op_{129}	iastore
op_{27}	fsub	op_{130}	monitorexit

Continued on next page

Table A.1 – continued from previous page

Elements	Opcodes	Elements	Opcodes
op_{28}	lmul	op_{131}	arraylength
op_{29}	putfield	op_{132}	f2i
op_{30}	ifne	op_{133}	nop
op_{31}	ifnonnull	op_{134}	isub
op_{32}	saload	op_{135}	f2l
op_{33}	invokespecial	op_{136}	aload_0
op_{34}	if_icmpne	op_{137}	astore_0
op_{35}	fcmpg	op_{138}	aload_1
op_{36}	ineg	op_{139}	goto
op_{37}	fstore	op_{140}	astore_1
op_{38}	fadd	op_{141}	aload_2
op_{39}	fcmpl	op_{142}	ldc_w
op_{40}	d2f	op_{143}	astore_2
op_{41}	idiv	op_{144}	laload
op_{42}	astore	op_{145}	aload_3
op_{43}	instanceof	op_{146}	astore_3
op_{44}	iflt	op_{147}	ireturn
op_{45}	bipush	op_{148}	land
op_{46}	if_icmplt	op_{149}	dstore_0
op_{47}	d2i	op_{150}	lcmp
op_{48}	invokestatic	op_{151}	dstore_1
op_{49}	invokevirtual	op_{152}	ixor
op_{50}	lor	op_{153}	drem
op_{51}	dup2_x1	op_{154}	dup2
op_{52}	lookupswitch	op_{155}	dstore_2

Continued on next page

Table A.1 – continued from previous page

Elements	Opcodes	Elements	Opcodes
op_{53}	d2l	op_{156}	dstore_3
op_{54}	dup2_x2	op_{157}	iadd
op_{55}	if_acmpeq	op_{158}	baload
op_{56}	i2b	op_{159}	iinc
op_{57}	freturn	op_{160}	fstore_0
op_{58}	i2c	op_{161}	lload_0
op_{59}	return	op_{162}	multianewarray
op_{60}	i2d	op_{163}	lrem
op_{61}	ret	op_{164}	fstore_1
op_{62}	i2f	op_{165}	lload_1
op_{63}	sastore	op_{166}	aconst_null
op_{64}	dload	op_{167}	fstore_2
op_{65}	lload	op_{168}	lload_2
op_{66}	dsub	op_{169}	dneg
op_{67}	fastore	op_{170}	fstore_3
op_{68}	ldc	op_{171}	lload_3
op_{69}	iaload	op_{172}	reserved
op_{70}	iload_0	op_{173}	lushr
op_{71}	i2l	op_{174}	istore_0
op_{72}	daload	op_{175}	lstore
op_{73}	iload_1	op_{176}	istore_1
op_{74}	iload_2	op_{177}	ddiv
op_{75}	iload_3	op_{178}	istore_2
op_{76}	lsub	op_{179}	ifnull
op_{77}	dreturn	op_{180}	wide

Continued on next page

Table A.1 – continued from previous page

Elements	Opcodes	Elements	Opcodes
op_{78}	dadd	op_{181}	swap
op_{79}	i2s	op_{182}	istore_3
op_{80}	bastore	op_{183}	lstore_0
op_{81}	lxor	op_{184}	lstore_1
op_{82}	imul	op_{185}	xxxunusedxxx1
op_{83}	dastore	op_{186}	lneg
op_{84}	new	op_{187}	fmul
op_{85}	ifge	op_{188}	castore
op_{86}	ladd	op_{189}	lstore_2
op_{87}	invokeinterface	op_{190}	lstore_3
op_{88}	if_icmpge	op_{191}	ldc2_w
op_{89}	iushr	op_{192}	ldiv
op_{90}	sipush	op_{193}	ishl
op_{91}	monitorenter	op_{194}	iconst_m1
op_{92}	ifeq	op_{195}	jsr_w
op_{93}	dconst_0	op_{196}	jsr
op_{94}	dconst_1	op_{197}	if_acmpne
op_{95}	if_icmpeq	op_{198}	pop2
op_{96}	tableswitch	op_{199}	faload
op_{97}	frem	op_{200}	areturn
op_{98}	dup_x1	op_{201}	l2d
op_{99}	newarray	op_{202}	ishr
op_{100}	dup_x2	op_{203}	aload
op_{101}	caload	op_{204}	breakpoint
op_{102}	fconst_0	op_{205}	l2f

Continued on next page

Table A.1 – continued from previous page

Elements	Opcodes	Elements	Opcodes
op_{103}	istore	op_{206}	iload

A.2.2 Collberg's Sample Set Information

Table A.2: Collberg's Sample Set

Serial Number	Program Name	size (in opcodes)
1	TTT	1208
2	web_a2cat11	18199
3	web_acme	8941
4	web_activation	5340
5	web_ActiveRegionExplorer	71723
6	web_aglight-pac	757
7	web_Agna_2_kit	1084
8	web_allkara-en	101474
9	web_allkara-x-en	101474
10	web_aocode-public.src	38768
11	web_aoserv-examples.src	871
12	web_Arachnophilia	79929
13	web_archery	3673
14	web_ArraySort2D1	275
15	web_aspectj-1.1.0	3807
16	web_AutoSim	13261
17	web_BattleShip	2100

Continued on next page

Table A.2 – continued from previous page

Serial Number	Program Name	size (in opcodes)
18	web_BBI	14305
19	web_BBIagent	49527
20	web_bigal	1388
21	web_biojava-1.00	45637
22	web_bluej-121	1840
23	web_bluej-122	1840
24	web_bluej-130beta2	1843
25	web_btools-1.1	5101
26	web_bytecode-0.90	4393
27	web_candy	2194
28	web_ccmb	210318
29	web_chimera	45390
30	web_ChronicleLite-bin-v1.2	25055
31	web_ClassMapper	85081
32	web_CodeProcessor	3474
33	web_commons-collections	22102
34	web_commons-pool	3805
35	web_connect	39
36	web_Conzilla	63154
37	web_Conzilla1.1Beta2	74153
38	web_cparser	4217
39	web_crimson	22625
40	web_crimson_mod	22759
41	web_CryptoHeaven	51064
42	web_CurveSimulator	33301

Continued on next page

Table A.2 – continued from previous page

Serial Number	Program Name	size (in opcodes)
43	web_customizer	15168
44	web_Cvt2Mae	40250
45	web_DecodeHtml	3731
46	web_desktop_indicator	257
47	web_devpgjdbc1	20157
48	web_devpgjdbc2	24767
49	web_devpgjdbc3	27137
50	web_DigestCalc	1439
51	web_DMLObjectModeler	29023
52	web_DocWiz0.68	58869
53	web_dom	21
54	web_dss	320
55	web_Dylan	12108
56	web_ecp1_0beta	7359
57	web_Edgeis	73561
58	web_EditFiles	1038
59	web_FetchFiles	2254
60	web_ff	1042
61	web_ffgui	1007
62	web_ffthis	3083
63	web_figue	35081
64	web_FileEdit	4709
65	web_FindFiles	358
66	web_flat	22003
67	web_form	6157

Continued on next page

Table A.2 – continued from previous page

Serial Number	Program Name	size (in opcodes)
68	web_foxhunt-0.4	2209
69	web_fuzzyide	12078
70	web_geometria	85312
71	web_geotransform	5594
72	web_gif	1201
73	web_grades	36659
74	web_graphpanel	2986
75	web_GrsFinder	954
76	web_HotEqn	10124
77	web_hqt	1040
78	web_HTMLEditorPro	59425
79	web_i503emulator	11309
80	web_ifxjdbc	133525
81	web_ij	131758
82	web_ImageWarper	1891
83	web_interface	11286
84	web_ITower	9925
85	web_jabadex	15472
86	web_JarCreator	806
87	web_jar-util	5118
88	web_jasminclasses-sable-1.2	27686
89	web_jasmin-sable-1.2	23867
90	web_java2html.vj0.2	3338
91	web_java-getopt-1.0.9	1654
92	web_javapopt	1624

Continued on next page

Table A.2 – continued from previous page

Serial Number	Program Name	size (in opcodes)
93	web_jaxp	1790
94	web_jaxp-api	1541
95	web_jblitz	395595
96	web_jcalc_dist	12226
97	web_jcchart401K	172732
98	web_jce1_2_1	21695
99	web_jcm1.0-config	44854
100	web_jconnect45	35753
101	web_jconnect55	40800
102	web_jdbc2_0-stdext	22
103	web_jdbc6.5-1.2	12653
104	web_jdbc7.0-1.1	12190
105	web_jdeps	22270
106	web_jdictionary	22989
107	web_jDvi	9375
108	web_JevaESUI2001-03-03	46948
109	web_Jexa1999-10-11	8961
110	web_jexn0.1.1	5583
111	web_jext-install	16583
112	web_jfinger-v0.05b	10264
113	web_jh	42521
114	web_jhbasic	25358
115	web_JJukeboxSetup	25438
116	web_JKeyboard	17051
117	web_jmix_dist	293

Continued on next page

Table A.2 – continued from previous page

Serial Number	Program Name	size (in opcodes)
118	web_jndi	7829
119	web_jode-1.0.93-1.2	71030
120	web_jode-1.1.1-JDK1.1	88489
121	web_jp0211jt	439
122	web_jpe	7707
123	web_jpp	33573
124	web_jraceman-0_3_8	1629
125	web_js	85545
126	web_js2	87954
127	web_jstyle	4813
128	web_junit	5079
129	web_Jupiter	2615
130	web_jV	14386
131	web_jV_src	14618
132	web_jxtasecurity	8363
133	web_ladder	8133
134	web_lava	41093
135	web_ldap	20406
136	web_LINK	536
137	web_linx	8274
138	web_linxbuilder	6505
139	web_live	23471
140	web_logan-games	13680
141	web_Logisim	21689
142	web_Lucifers	318

Continued on next page

Table A.2 – continued from previous page

Serial Number	Program Name	size (in opcodes)
143	web_Lunar	826
144	web_LunarHeights	1983
145	web_LV154	24734
146	web_m_date_entry	13229
147	web_m2mpapi20010504	3855
148	web_m2mpapi20011221	3952
149	web_mad	13108
150	web_MAExplorer	158474
151	web_Marquee	974
152	web_Mars	2010
153	web_mindterm	72109
154	web_MMLViewerApplet	137634
155	web_ModEdit	7346
156	web_moses	361496
157	web_moses100install	44119
158	web_mysql	41101
159	web_mysql-connector-java-2.0.14-bin	12852
160	web_myxml-1.3	685
161	web_nanoxml-2.2.3	5355
162	web_nanoxml-lite-2.2.3	1812
163	web_Navigation	527
164	web_networkbk_client	22224
165	web_networkbk_server	22224
166	web_netx	12191
167	web_nis	12282

Continued on next page

Table A.2 – continued from previous page

Serial Number	Program Name	size (in opcodes)
168	web_or124	72965
169	web_oracle	221320
170	web_oro	13967
171	web_pac3d	161503
172	web_parser	3094
173	web_ParTasks-0.1.0	871
174	web_patbinfree153	32378
175	web_pg72jdbc1	15154
176	web_pg73jdbc2ee	26921
177	web_pg73jdbc3	27366
178	web_photoindex_dist	3689
179	web_photoindex3_dist	5372
180	web_ping_icmp	196
181	web_pircbot	3568
182	web_PlanetFinder	897
183	web_PopsEdit	30378
184	web_Posse	41899
185	web_postgresql	27385
186	web_powerforms	38072
187	web_ProblemParser	1805
188	web_profiler	16327
189	web_providerutil	11377
190	web_proxy	7882
191	web_PSOL	58945
192	web_ptah	1540

Continued on next page

Table A.2 – continued from previous page

Serial Number	Program Name	size (in opcodes)
193	web_qoca-1.0beta2-mod	76393
194	web_Rangavalli-1.2	53022
195	web_rockz	41483
196	web_RollOver	962
197	web_run	4223
198	web_sax	2995
199	web_scoreboard	3614
200	web_Scramble	1768
201	web_SeqSpace	96682
202	web_shared	18255
203	web_skinlf	44270
204	web_smpp	16012
205	web_SpidersRUs	40340
206	web_splat	1728
207	web_stockClient	590
208	web_stockServer	780
209	web_sunjce_provider	33518
210	web_suntimes	834
211	web_sxp	28779
212	web_Tank	13114
213	web_template	1328
214	web_TextScroller	1313
215	web_toy_1.4	60418
216	web_ttt2	3364
217	web_turtletracks	22307

Continued on next page

Table A.2 – continued from previous page

Serial Number	Program Name	size (in opcodes)
218	web_TZTester	514
219	web_unit-util	1953
220	web_UofAC114	104
221	web_utils	28891
222	web_vienna	15474
223	web_virbotchi	17444
224	web_WiynRo	63000
225	web_WP	4307
226	web_xalan	127515
227	web_xlink	653
228	web_xml	10409
229	web_xmlbench	4277
230	web_xmlser-rc5	6152
231	web_XmlWriter	4666
232	web_xpointer	9811
233	web_XsltEditor	4331
234	web_YMStrings	539

A.2.3 Collberg's Sample Set Clearance

In experiment C, we assume that the attacker is lazy and will not clean his sample set. In experiment set A, we assume that the attacker is very skeptical. He/she will check programs which are suspiciously to be related to each other. For each group of programs suspiciously to be related, he/she will randomly select one program and abandon all other programs in that group. Thus he/she can increase the accuracy of his classifier.

The removal is based on the observations of the attacker. The serial number of programs to be removed from the Collberg's Sample set are {19, 37, 9, 22, 24, 33, 40, 47, 48, 88, 94, 100, 103, 114, 119, 125, 138, 147, 157, 159, 162, 164, 175, 176, 177, 178, 207, 229, 230, 129, 152, 144, 75, 143, 182 }. So totally thirty-five programs are removed.

It is possible that the attacker will remove some programs actually unrelated. However, such removal will affect only the total number of samples. While no restriction on the number of unwatermarked programs the attacker can obtain, such removal will not be a problem for the attacker.

A.2.4 Watermark Embedding Procedure

The watermark embedding is done by using Sandmark. To embed a C-T watermark into a program by Sandmark, three steps are necessary : *annotating*, *tracing*, and *embedding*.

In the *annotating* step, we execute the following command to annotate the class file of the program to be watermarked. (That class file must contain a *main()* function) :

java -cp bcel.jar;sandmark.jar; CTifier ⟨ class to be annotated ⟩

To run the above command, we should keep the four files (CTifier, the class file to be annotated, bcel.jar and sandmark.jar) in the same folder.

Then we can use the annotated class to replace its original version and pack the program into a jar file as follows:

jar cvfm ⟨ target jar file name ⟩ man.txt *.*

In the *tracing* step, we select the *Dynamic Watermark* tab in Sandmark interface. Next in *Dynamic Watermark* panel, we select *Collberg/Thomborson* in the *algorithm option* box. Then we select the *Trace* tab. Next we select the program to be watermarked from the *Input File* text field. In the *Main Class* text field, the name of the mainclass (the class with the *main()* function) of the program to be watermarked must be entered. Then we click on *Start*. At this point, the program will be running. We make some inputs (e.g.input by clicking buttons with mouse) as *key*, and then close the program. Lastly, we click *Done*.

In the *embedding* step, we select the *embed* tab in the *Dynamic Watermark* panel of Sandmark. Then we input our watermark w in the *Watermark* text field. In watermark embedding, there are many parameters can be selected in the *Embed* panel. we will change two check boxes (*Numeric Watermark*, *use Cycle Graph*) for each watermark embedding procedure while keep all other parameters as their default value.

The default values of parameters fixed in our experiment are as follows:

- Storage Policy: 'root'
- Storage Method: 'array:vector:hash'
- Storage Location: 'formal'
- Protection Method: 'if:safe:try'
- Graph Type: '*'
- Subgraph Count: '2'
- dump Intermediate Code: 'not select'
- Inline Code: our choice: 'not select'
- Replace Watermark Class: 'not select'
- Dump Intermediate Code: 'not select'

The key set used in our experiment is key_E. Table A.3 lists the elements of key_E.

The watermark set used in our experiment is $\mathcal{W}_E$. Table A.4 lists the elements of $\mathcal{W}_E$.

Table A.4: Watermark Set for Experiment

Elements	**Watermarks**	**Elements**	**Watermarks**
w_1	102708082	w_{1001}	ko234asx
w_5	386872058	w_{1005}	rusem
Continued on next page			

Table A.4 – continued from previous page

Elements	Watermarks	Elements	Watermarks
w_{15}	911393057	w_{1015}	45gtr
w_{16}	123564098	w_{1016}	jpsrty
w_{18}	14565167	w_{1018}	jitih
w_{30}	474080610	w_{1030}	gi990lu
w_{50}	673096763	w_{1050}	dtghjg
w_{54}	111042854	w_{1054}	dss
w_{60}	184208742	w_{1060}	erserwerty
w_{69}	468498932	w_{1069}	681
w_{73}	605378825	w_{1073}	ojko
w_{77}	949414928	w_{1077}	euio
w_{78}	986551407	w_{1078}	v45
w_{87}	858354009	w_{1087}	awe763
w_{92}	122682574	w_{1092}	xssdf
w_{123}	710435026	w_{1123}	uilj
w_{141}	983396868	w_{1141}	484m8h
w_{155}	542323271	w_{1155}	4er0spo
w_{197}	308208724	w_{1197}	co8p
w_{205}	4056306	w_{1205}	trtrh
w_{234}	941246940	w_{1234}	vb5bk8

A.2.5 Pattern Retrieve Procedure

The pattern retrieve procedure is shown in Figure 4.1. Details are as follows:

Firstly, under Unix, we use the "sh OPTfilepretest" command to disassemble Java class files and retrieve the $k-gram$ sequence from a Java program.

keys	INPUT
key_1	Click ``upleft square"=>click ``middle square"=>click ``downright square"=>exit
key_5	Click ``OK"
key_{15}	Click``next"=>``cancel"=>``Yes"
key_{16}	Click ``file"=>``new"=>``deterministic automation"=>``no"=>``quit"
key_{18}	Click``next"=>``close"
key_{30}	in ``user name" field, input ``tt" => in ``password" field, input ``123"=> Click ``options"=>select all checkbox-> click ``OK"
key_{50}	(do nothing)
key_{54}	Click ``find DSS image"=>Close program
key_{60}	(do nothing)
key_{69}	(do nothing)
key_{73}	Click ``file"=>click ``new"=> in ``year" field, input ``1111"=> in ``Course" field, input ``1111"=>in ``description' field, input ``1111"=>click ``OK"=>click ``file"=> click ``exit"
key_{77}	(do nothing)
key_{78}	Click ``file"=>click ``exit"=>click ``no"
key_{87}	Click ``file"=>click ``New"=>click ``Jar"=>click ``Open"=>click ``Cancel"=>click ``file"=>click ``exit"
key_{92}	(do nothing)
key_{123}	(do nothing)
key_{141}	Click ``file"=>click ``new"=>click ``Yes"=>click ``file"=> click ``quit"=> click ``no"
key_{155}	Click ``file"=>click ``exit"
key_{197}	Click ``close"
key_{205}	Click ``OK"=>click ``file"=>click ``new"=>click ``cancel"
key_{234}	(do nothing)

Table A.3: Elements of keys used in our experiment

"OPTfilepretest" is a shell program that we developed. "OPTfilepretest" uses a tool provided by JDK called "javap" to disassemble Java class files. The command used by "OPTfilepretest" is "disassemble javap -c -l -private < classfile >"

Secondly, we use "sh OPTdatacollection" followed by the "sh OPTruntransform" command to retrieve the pattern vectors. The pattern vectors are saved in a .csv file which can be read by SPSS.

A.2.6 SPSS Analysis Procedure

In Experiment set A, the syntax of discriminant analysis used by $run_i, (1 \leq i \leq 9)$ is as follows:

DISCRIMINANT /GROUPS=category(1 2) /VARIABLES=dcmpg l2i aastore dcmpl dload_0 dload_1 fload_0 dload_2 fload_1 iand dload_3 fload_2 fload_3 lreturn impdep1 athrow impdep2 dmul ifle lastore aaload anewarray if_icmple irem pop checkcast fsub lmul putfield ifne ifnonnull saload invokespecial if_icmpne fcmpg ineg fstore fadd fcmpl d2f idiv astore instanceof iflt bipush if_icmplt d2i invokestatic invokevirtual lor dup2_x1 lookupswitch d2l dup2_x2 if_acmpeq i2b freturn i2c return i2d

ret i2f sastore dload lload dsub fastore ldc iaload iload_0 i2l daload iload_1 iload_2 iload_3 lsub dreturn dadd i2s bastore lxor imul dastore new ifge ladd invokeinterface if_icmpge iushr sipush monitorenter ifeq dconst_0 dconst_1 if_icmpeq tableswitch frem dup_x1 newarray dup_x2 caload fconst_0 istore fconst_1 ifgt fconst_2 ior goto_w putstatic if_icmpgt dstore iconst_0 lshl iconst_1 iconst_2 getfield fneg iconst_3 fload lconst_0 iconst_4 f2d dup lconst_1 iconst_5 getstatic fdiv lshr iastore monitorexit arraylength f2i nop isub f2l aload_0 astore_0 aload_1 goto astore_1 aload_2 ldc_w astore_2 laload aload_3 astore_3 ireturn land dstore_0 lcmp dstore_1 ixor drem dup2 dstore_2 dstore_3 iadd baload iinc fstore_0 lload_0 multianewarray lrem fstore_1 lload_1 aconst_null fstore_2 lload_2 dneg fstore_3 lload_3 reserved lushr istore_0 lstore istore_1 ddiv istore_2 ifnull wide swap istore_3 lstore_0 lstore_1 xxxunusedxxx1 lneg fmul castore lstore_2 lstore_3 ldc2_w ldiv ishl iconst_m1 jsr_w jsr if_acmpne pop2 faload areturn l2d ishr aload breakpoint l2f iload /ANALYSIS ALL /PRIORS EQUAL /STATISTICS=MEAN STDDEV UNIVF BOXM COEFF RAW CORR COV GCOV TCOV TABLE /PLOT=CASES /CLASSIFY=NONMISSING SEPARATE .

In Experiment C, the syntaxes of discriminant analysis are as follows:

1. run_{10} of Experiment C

   DISCRIMINANT /GROUPS=Category(1 2) /VARIABLES=aastore iand aaload anewarray pop if_icmple checkcast putfield ifne invokespecial if_icmpne astore bipush iflt if_icmplt invokestatic invokevirtual if_acmpeq return lload ldc i2l iload_1 iload_2 iload_3 new sipush ifeq istore ior iconst_0 putstatic iconst_1 iconst_2 getfield iconst_3 iconst_4 dup getstatic nop isub aload_0 aload_1 astore_0 aload_2 astore_1 goto astore_2 aload_3 astore_3 ireturn lcmp dup2 iadd iinc lstore istore_1 istore_2 istore_3 ldc2_w iconst_m1 if_acmpne pop2 areturn aload iload /ANALYSIS ALL /METHOD=MAHAL /PIN= .05 /POUT= .10 /PRIORS EQUAL /HISTORY /STATISTICS=MEAN STDDEV UNIVF BOXM COEFF RAW CORR COV GCOV TCOV /PLOT=CASES /CLASSIFY=NONMISSING SEPARATE .

2. run_{11} of Experiment C

   (Notice: A.B is a 2-gram where A is the first opcode and B is the second opcode in the 2-gram)

   DISCRIMINANT /GROUPS=Category(1 2) /VARIABLES=bipush.invokespecial if_icmplt.iconst_0 iconst_1.istore_1 iconst_3.invokespecial iload_1.aaload aload_1.invokevirtual astore_0.iconst_m1 if_icmplt.nop goto.aload_0 istore_1.iconst_0 aload_0.invokespecial lstore.pop2 if_icmpne.iconst_1 if_acmpne.iconst_1 iload_1.iconst_1 aload_2.invokevirtual iload_1.iconst_2 lload.ldc2_w nop.goto goto.ldc iload_3.aaload invokevirtual.ifne ldc.invokevirtual iconst_3.iadd getfield.iconst_1 iconst_2.iconst_4

iand.istore_1 aload_3.goto getfield.iconst_2 astore.nop astore_1.aload_1 if_icmplt.iload_1 getfield.getfield iload_2.aaload istore_2.iload_2 invokevirtual.istore_2 iadd.iload_2 iadd.iload_3 istore_1.getstatic aload_0.new sipush.aload aload_0.invokevirtual ldc.invokespecial invokestatic.goto invokevirtual.ireturn iload_1.bipush astore_3.aload_3 iadd.aaload getstatic.invokevirtual istore_1.iinc astore_2.aload_0 invokevirtual.if_acmpne aload.invokevirtual aaload.iload_1 aaload.invokevirtual aload.bipush iload_3.bipush getstatic.invokespecial goto.new if_icmple.iconst_m1 pop.iinc bipush.aload invokevirtual.aload_0 invokevirtual.aload_1 invokestatic.return iload_2.bipush dup.getstatic getfield.aload_0 getfield.ifne invokevirtual.nop getfield.ireturn iflt.iload_2 invokespecial.astore_1 iconst_2.if_icmpne ldc.areturn invokespecial.aload_0 iconst_2.iadd getstatic.iload_2 iconst_2.istore_1 invokespecial.aload_2 iconst_1.goto iload_1.invokestatic getfield.astore iconst_1.putfield ldc.iconst_1 getfield.getstatic iconst_0.invokestatic aload_1.putfield ireturn.new ldc.if_acmpne astore_2.iconst_0 iconst_3.iconst_3 invokevirtual.checkcast sipush.invokevirtual goto.iconst_0 if_icmplt.return aload.getfield goto.iconst_2 dup.ldc ifeq.aload_0 dup2.lstore pop2.aload_0 iconst_0.iconst_4 sipush.invokespecial if_icmpne.ldc iconst_1.isub ior.istore_1 pop2.lload invokevirtual.ifeq pop.return goto.nop ldc.astore_1 if_icmplt.aload_0 istore_2.aload_0 iload.iload_3 iload_1.i2l if_icmpne.aload_0 iload_2.iconst_1 iload_2.iconst_2 iload_1.invokevirtual invokevirtual.areturn iload_2.iconst_3 goto.astore_0 getstatic.getstatic pop.aload iconst_1.iadd ifne.iconst_1 aload_0.iconst_0 ifeq.goto aload_0.iconst_1 iload_2.invokevirtual invokevirtual.goto putfield.iconst_0 aload_0.iconst_2 iinc.iload_1 invokespecial.ldc dup.new aload_0.getfield if_acmpeq.aload_0 iinc.iload_2 iinc.iload_3 checkcast.astore nop.new invokevirtual.if_icmpne istore_3.goto invokestatic.iconst_1 iinc.iload putfield.aload anewarray.putfield iconst_1.bipush putfield.new iload_1.new invokevirtual.aload ifeq.getstatic istore.goto iand.ireturn iload_2.iflt putfield.return iconst_4.bipush aload_0.getstatic nop.aload aload.sipush aload_0.bipush invokevirtual.invokespecial iload_3.iload_3 iconst_0.istore_1 iconst_2.putfield iconst_0.istore_2 invokevirtual.return iconst_0.istore_3 getfield.ldc dup.iconst_3 pop.nop iload_2.iload_2 ifne.aload_0 invokespecial.invokevirtual invokestatic.invokevirtual iand.aload_0 ifne.aload_2 sipush.sipush aload_2.nop getfield.i2l ifeq.iconst_0 bipush.if_icmple bipush.anewarray putstatic.getstatic aaload.ldc bipush.iadd getstatic.ireturn iconst_3.if_icmplt invokevirtual.invokevirtual getfield.invokevirtual dup.sipush iadd.invokespecial ldc.goto ldc.if_acmpeq invokespecial.invokespecial goto.iload_1 aastore.iinc invokevirtual.pop getstatic.ldc iconst_0.aload_0 invokespecial.aastore invokespecial.putfield invokespecial.return invokevirtual.iinc aload_0.iload_1 isub.istore_2 aload_0.iload_2 nop.return new.dup aload.nop putfield.goto aload_0.iload_3 putfield.aload_0 pop.aload_0 getfield.new invokespecial.putstatic istore_2.goto iconst_0.ireturn iload_3.iconst_1 nop.nop lcmp.ifne invokespecial.astore iload_3.iconst_2 if_acmpne.aload_0 iload_3.iconst_3 nop.astore putfield.nop invokespecial.ior bipush.if_icmplt getfield.iload putstatic.return aload.putfield aaload.new i2l.dup2 if_icmplt.new astore.aload invokestatic.getstatic iconst_1.if_icmpne getstatic.new invokevirtual.putstatic goto.getstatic dup.aload_0 dup.aload_1 invokevirtual.iconst_0 aload_0.sipush goto.pop

putstatic.goto aastore.aload_0 astore.new if_icmpne.getstatic invokevirtual.getstatic invokevirtual.iload_1 iload_1.aload_0 getfield.iload_1 aload_1.astore_2 iload_1.aload_2 getfield.iload_2 ldc.aload_0 iconst_0.putfield iconst_m1.invokestatic getfield.iload_3 invokevirtual.ldc iload_3.iadd astore_2.iinc iload_1.ireturn aload_0.aload_0 iconst_0.istore nop.lload aload_0.aload_1 iload.iconst_3 ifne.ldc bipush.invokevirtual iconst_1.invokespecial dup.invokespecial istore_1.goto invokevirtual.invokestatic nop.aload_0 aload.aload nop.astore_2 ldc2_w.lcmp nop.astore_3 iconst_0.iand invokevirtual.astore_2 aload_0.ldc i2l.pop2 /ANALYSIS ALL /METHOD=MAHAL /PIN= .05 /POUT= .10 /PRIORS EQUAL /HISTORY /STATISTICS=MEAN STDDEV UNIVF BOXM COEFF RAW CORR COV GCOV TCOV /PLOT=CASES /CLASSIFY=NONMISSING SEPARATE .

3. run_{12} of Experiment C

   (Notice: A.B.C is a 3-gram where A is the first opcode, B is the second opcode and C is the third opcode in the 3-gram)

DISCRIMINANT /GROUPS=Category(1 2) /VARIABLES=iand.istore_1.iinc dup.iconst_3.iconst_3 ldc.invokevirtual.iconst_0 istore_3.goto.iload_1 iconst_3.iadd.iload_2 iconst_3.iadd.iload_3 if_icmple.iconst_m1.invokestatic istore_1.iinc.iload_2 istore_1.iinc.iload_3 iand.aload_0.getfield sipush.sipush.invokevirtual goto.aload_0.invokevirtual ldc.invokespecial.putfield if_acmpne.iconst_1.istore_1 istore_2.goto.iload_1 bipush.invokespecial.ior aload.putfield.nop if_acmpne.iconst_1.goto i2l.dup2.lstore anewarray.putfield.iconst_0 invokevirtual.invokevirtual.getstatic getstatic.iload_2.invokevirtual ifne.aload_2.nop goto.nop.nop nop.aload_0.getfield invokevirtual.invokevirtual.aload_0 aastore.aload_0.getfield aload_0.iconst_2.putfield astore_2.iinc.iload goto.getstatic.ldc iload_2.aaload.invokevirtual aload_0.invokevirtual.ifeq iinc.iload_2.iconst_3 putfield.goto.pop ldc.if_acmpne.aload_0 iconst_0.iand.istore_1 if_acmpne.aload_0.getfield invokevirtual.ifeq.aload_0 iconst_0.iconst_4.bipush iload_2.bipush.iadd lload.ldc2_w.lcmp invokevirtual.ifne.iconst_1 invokevirtual.pop.aload putstatic.goto.astore_0 aload_0.iload_2.invokevirtual dup.ldc.invokespecial putstatic.getstatic.ldc aload_0.getfield.getstatic invokevirtual.invokevirtual.iconst_0 getfield.iload_3.aaload aaload.invokevirtual.ldc istore.goto.new iconst_3.if_icmplt.iload_1 getstatic.getstatic.invokevirtual dup.getstatic.invokespecial iconst_2.if_icmpne.iconst_1 invokevirtual.putstatic.getstatic invokevirtual.aload_1.invokevirtual ldc.invokevirtual.aload_0 aload_0.iconst_0.iconst_4 astore_2.iconst_0.istore_3 astore.aload.getfield invokevirtual.aload_0.ldc dup.invokespecial.aload_2 aload.sipush.aload getstatic.invokevirtual.iconst_0 iconst_1.iadd.iload_2 iconst_1.iadd.iload_3 aload_0.aload_0.getfield pop2.aload_0.iload_1 invokevirtual.aload_0.getfield invokevirtual.invokevirtual.ldc ifne.aload_0.invokevirtual goto.aload_0.getstatic new.dup.new invokespecial.invokevirtual.nop invokevirtual.ifeq.iconst_0 iconst_0.istore_2.goto invokespecial.astore.aload invokevirtual.if_acmpne.aload_0 iload_3.iconst_2.iadd nop.new.dup get-

field.invokevirtual.ldc aaload.invokevirtual.aload_0 aaload.invokevirtual.pop bipush.if_icmplt.new aload_1.invokevirtual.invokespecial

aload_2.nop.astore getstatic.invokevirtual.invokespecial iload_1.new.dup iconst_1.isub.istore_2 invokespecial.invokespecial.putstatic

aaload.ldc.invokevirtual iadd.iload_3.bipush aload_0.iconst_1.putfield iload_2.invokevirtual.ifne ldc.astore_1.aload_1 dup.invokespecial.astore_1

iconst_3.iconst_3.invokespecial invokevirtual.iload_1.i2l

iload_2.iconst_2.iadd aload_0.getfield.invokevirtual ldc.if_acmpne.iconst_1 astore.aload.aload pop.iinc.iload_1 iconst_2.iconst_4.bipush

aload_0.iload_1.invokevirtual invokevirtual.aload_0.sipush aload.getfield.astore iload_1.aaload.ldc istore_1.getstatic.new

bipush.invokevirtual.checkcast aload_2.invokevirtual.aload_0 aload_2.invokevirtual.aload_1 dup.new.dup aaload.new.dup

invokespecial.invokevirtual.istore_2 iconst_1.istore_1.iconst_0 iinc.iload_3.bipush ldc.invokespecial.aastore invokevir-

tual.ldc.invokevirtual aload.bipush.aload invokevirtual.checkcast.astore iconst_2.iadd.invokespecial invokevirtual.goto.aload_0

invokevirtual.aload_0.invokevirtual aload_0.iload_1.aload_2 aload.aload.putfield getfield.invokevirtual.pop if_icmpne.getstatic.iload_2

putfield.aload_0.iconst_2 iconst_1.invokespecial.putfield goto.iconst_2.putfield aload.sipush.invokevirtual getfield.ldc.invokevirtual

aload.bipush.invokevirtual invokevirtual.pop.aload_0 aload_0.invokevirtual.ifne aload_0.bipush.anewarray ifne.ldc.goto

i2l.pop2.aload_0 invokevirtual.aload_0.getstatic astore_2.aload_0.invokevirtual astore.aload.sipush invokevirtual.iload_1.invokevirtual

iload_2.iconst_3.if_icmplt getfield.iload_2.invokevirtual aload_0.getfield.i2l astore_1.aload_1.astore_2 getfield.iconst_2.if_icmpne

ldc.invokevirtual.invokevirtual invokespecial.aload_0.new ldc.invokevirtual.iinc iconst_3.invokespecial.invokevirtual in-

vokespecial.ldc.invokevirtual getfield.iload_2.aaload invokespecial.astore.new aload_0.getstatic.invokevirtual invoke-

virtual.ifeq.goto goto.iconst_0.aload_0 goto.iconst_2.istore_1 ifeq.iconst_0.ireturn iconst_1.istore_1.goto iload_1.invokevirtual.aload_0

goto.iload_1.aload_0 invokespecial.ior.istore_1 aload_0.invokespecial.aload_0 invokevirtual.astore_2.iconst_0 new.dup.aload_0

if_icmpne.iconst_1.istore_1 new.dup.aload_1 aload_0.getfield.iload iload_3.bipush.if_icmplt aload_0.sipush.sipush ldc.invokevirtual.goto

ldc.if_acmpeq.aload_0 iload_1.aload_2.invokevirtual invokevirtual.putstatic.goto iload_3.iload_3.iconst_1 invokevirtual.nop.aload_0

ior.istore_1.iconst_0 iload_3.iload_3.iconst_3 iconst_0.istore_1.goto iload_2.invokevirtual.getstatic invokevirtual.istore_2.aload_0

if_icmplt.nop.aload_0 iconst_1.istore_1.iinc iload_3.iconst_1.iadd getfield.getfield.ldc if_icmpne.iconst_1.goto nop.lload.ldc2_w

sipush.invokevirtual.aload invokevirtual.invokespecial.invokevirtual iconst_0.aload_0.getfield invokespecial.aastore.aload_0

lcmp.ifne.aload_2 lstore.pop2.lload invokespecial.invokevirtual.putstatic getfield.iload_1.new new.dup.sipush getstatic.invokevirtual.getstatic

bipush.iadd.invokespecial aload_0.getfield.ifne invokevirtual.iinc.iload_1 astore.nop.aload invokevirtual.iinc.iload_2 if_icmplt.iconst_0.istore_3

dup.invokespecial.invokevirtual getstatic.invokevirtual.goto iload_2.iconst_1.iadd iload.iconst_3.if_icmplt astore.new.dup

invokevirtual.pop.iinc ldc.invokevirtual.ifeq invokestatic.getstatic.iload_2 iload_3.aaload.invokevirtual astore_1.aload_1.invokevirtual

iload_2.bipush.if_icmple if_icmplt.aload_0.getfield dup.getstatic.invokevirtual aload_0.getfield.iconst_1 aload_0.getfield.iconst_2

aload_0.iconst_0.putfield pop.aload_0.ldc aload_1.astore_2.iconst_0 iload_2.bipush.if_icmplt getfield.iload.iload_3 invoke-

virtual.goto.getstatic getfield.getstatic.invokevirtual putfield.aload.aload bipush.if_icmplt.aload_0 invokevirtual.ldc.if_acmpeq

iinc.iload_2.bipush aaload.invokevirtual.invokevirtual iconst_0.iand.aload_0 iflt.iload_2.bipush goto.getstatic.iload_2

putfield.nop.goto ldc.aload_0.getfield invokevirtual.if_acmpne.iconst_1 iload_1.invokestatic.invokevirtual ifeq.getstatic.ldc

goto.iload_1.iconst_1 aload_0.getfield.aload_0 goto.iload_1.iconst_2 getstatic.invokevirtual.ifeq dup2.lstore.pop2 nop.nop.goto

iload_3.iconst_3.if_icmplt pop.nop.nop aload.putfield.new ldc.invokevirtual.iload_1 nop.astore.new getfield.ifne.ldc bi-

push.aload.invokevirtual aload_0.getfield.ldc aload_1.invokevirtual.invokevirtual getstatic.invokevirtual.invokevirtual

aload_1.invokevirtual.aload_1 iadd.iload_2.bipush goto.new.dup nop.nop.lload aload.invokevirtual.pop invokevirtual.aload_0.new

putfield.iconst_0.istore_1 aload_0.iload_2.iload_2 checkcast.astore.nop invokevirtual.goto.iload_1 dup.aload_0.invokespecial

iload_1.bipush.if_icmplt invokevirtual.getstatic.getstatic bipush.anewarray.putfield goto.pop.nop iload_3.bipush.iadd

iload_1.iconst_1.if_icmpne invokevirtual.getstatic.invokevirtual iload_1.aload_0.iload_2 iload_1.aload_0.iload_3 getfield.invokevirtual.iconst_0

getstatic.invokevirtual.putstatic astore.aload.bipush iconst_1.goto.iconst_0 iconst_1.goto.iconst_2 iconst_2.putfield.aload_0

iconst_1.if_icmpne.getstatic getstatic.ldc.invokevirtual pop.aload.aload iload_2.iflt.iload_2 aload.putfield.aload put-

field.aload_0.iload_1 invokevirtual.aload.nop nop.aload_0.getstatic iconst_0.ireturn.new bipush.invokespecial.aload_0

bipush.invokespecial.invokevirtual ldc2_w.lcmp.ifne iload_1.aaload.new iconst_0.istore_1.iconst_0 iadd.iload_3.iconst_2

aload_0.iload_3.iload_3 bipush.if_icmplt.nop ldc.goto.ldc istore_2.iload_2.iflt new.dup.invokespecial anewarray.putfield.aload_0

invokevirtual.invokevirtual.goto aload.putfield.goto pop.aload_0.getfield iadd.invokespecial.ior getfield.iload_1.aaload

ior.istore_1.iinc astore_2.iinc.iload_3 invokestatic.iconst_1.isub getstatic.invokevirtual.aload_0 aload_0.iconst_2.iconst_4

new.dup.iconst_3 goto.aload_0.getfield istore_2.goto.aload_0 iconst_4.bipush.invokespecial getfield.i2l.pop2 istore_1.goto.aload_0

new.dup.ldc dup.invokespecial.putstatic nop.astore_3.aload_3 aload_2.invokevirtual.ldc iconst_2.if_icmpne.ldc nop.astore_2.aload_0

invokevirtual.ldc.if_acmpne dup.invokespecial.putfield ifne.iconst_1.goto pop2.lload.ldc2_w ifeq.goto.getstatic putfield.aload_0.getfield

aload_0.new.dup ldc.goto.aload_0 aload.nop.astore_3 getfield.getfield.iload_2 iload_1.aaload.invokevirtual aload_0.aload_1.putfield

if_icmpne.ldc.goto dup.invokespecial.aastore invokespecial.putfield.aload_0 iload_2.iload_2.iconst_1 iload_2.iload_2.iconst_3

aload_3.goto.nop nop.aload.getfield sipush.invokespecial.astore astore_3.aload_3.goto invokestatic.getstatic.invokevirtual

goto.iconst_0.iand iconst_1.if_icmpne.ldc ldc.iconst_1.bipush dup.aload_1.invokevirtual iinc.iload_3.iconst_3 invokespe-

cial.invokevirtual.aload_0 iconst_0.invokestatic.getstatic invokevirtual.invokestatic.iconst_1 astore_2.iconst_0.istore si-

push.invokevirtual.aload_0 iconst_1.bipush.invokespecial bipush.if_icmplt.iload_1 aastore.iinc.iload_1 invokespecial.aastore.iinc

istore_2.aload_0.getfield iconst_3.if_icmplt.new ifeq.aload_0.getstatic getstatic.invokespecial.invokespecial iconst_2.istore_1.iconst_0

aload_0.getfield.iload_1 aload_0.getfield.iload_2 aload_0.getfield.iload_3 invokevirtual.ifne.aload_0 aload_0.ldc.aload_0

getfield.astore.aload aaload.invokevirtual.if_icmpne iload_1.i2l.dup2 iinc.iload_1.bipush putfield.aload.bipush astore_0.iconst_m1.invokestatic

iload.iload_3.iadd goto.nop.astore_2 iadd.aaload.invokevirtual ldc.iconst_1.invokespecial putfield.aload_0.bipush aaload.iload_1.invokestatic if_acmpeq.aload_0.getfield aaload.invokevirtual.if_acmpne iconst_2.istore_1.getstatic iconst_m1.invokestatic.getstatic putfield.aload_0.new iload_1.invokevirtual.ifne invokevirtual.invokevirtual.invokevirtual ldc.invokevirtual.invokestatic if_icmplt.new.dup iload_1.aaload.iload_1 dup.invokespecial.ldc ireturn.new.dup invokespecial.aload_0.iconst_2 iload_2.invokevirtual.goto invokevirtual.iconst_0.invokestatic iload_3.iadd.aaload istore_1.iconst_0.istore_2 invokevirtual.invokevirtual.astore_2 nop.goto.nop iconst_0.istore.goto iconst_0.istore_3.goto dup.ldc.iconst_1 invokevirtual.iconst_0.istore_1 iload_1.invokevirtual.ldc invokevirtual.iconst_0.istore_2 invokespecial.aload_2.invokevirtual iload_3.iconst_3.iadd getstatic.new.dup invokevirtual.ifeq.getstatic iadd.iload_2.iconst_2 iload_2.aaload.ldc bipush.if_icmple.iconst_m1 getfield.iconst_1.if_icmpne iconst_1.if_icmpne.aload_0 new.dup.getstatic iinc.iload.iconst_3 aload_0.getfield.getfield bipush.if_icmplt.iconst_0 getfield.aload_0.getfield invokestatic.invokevirtual.aload_0 if_icmpne.aload_0.getfield ifeq.getstatic.new aload_0.invokespecial.invokevirtual getfield.new.dup invokespecial.aload_0.bipush putfield.new.dup istore_3.goto.new sipush.aload.invokevirtual invokevirtual.if_icmpne.iconst_1 invokevirtual.astore_2.iinc invokespecial.astore_1.aload_1 istore_1.goto.iconst_2 iload_2.invokevirtual.ifeq invokespecial.aload_0.aload_1 iload_2.iconst_3.iadd iload_1.iconst_2.if_icmpne aload_0.getfield.new putfield.aload.getfield dup.sipush.invokespecial isub.istore_2.iload_2 dup.invokespecial.astore iconst_3.if_icmplt.iconst_0 goto.astore_0.iconst_m1 ifeq.aload_0.aload_0 /ANALYSIS ALL /METHOD=MAHAL /PIN= .05 /POUT= .10 /PRIORS EQUAL /HISTORY /STATISTICS=MEAN STDDEV UNIVF BOXM COEFF RAW CORR COV GCOV TCOV /PLOT=CASES /CLASSIFY=NONMISSING SEPARATE .

This is the end!

Bibliography

[1] R. J. Anderson and Peticolas. On the limits of steganography. *IEEE Journal of Selected Areas in Communications*, 1998.

[2] Mauro Barni, Franco Bartolini, and Teddy Furon. A general framework for robust watermarking security. *Signal Processing*, 83(10):2069–2084, 2003.

[3] B.D.Ripley. *Pattern recognition and networks.* Cambridge university press, 1996.

[4] A.Z. Broder. On the resemblance and containment of documents. *Compression and Complexity of Sequences*, pages 21 – 29, June 1997.

[5] R. Chandramouli and N. Memon. How many pixels to watermark, 2000.

[6] Gu Y. Johnson H. Chow, S. and V Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Information Security: Fourth International Conference (ISC 2001)*, 2001.

[7] Collberg and Townsend. *Sandmark 3.4.0 help file.*

[8] C. Collberg and C. Thomborson. the limits of software watermarking, 1998.

[9] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, pages 311–324, 1999.

[10] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, volume 28, pages 735–746, August 2002.

[11] Christian S. Collberg, Clark Thomborson, and Gregg M. Townsend. Dynamic graph-based software fingerprinting. Submitted to TOPLAS in 15 April 2006, 62 pages.

[12] Ingemar J. Cox, Matthew L. Miller, and Jeffrey A. Bloom. *Digital watermarking*. Morgan Kaufmann, 2002.

[13] Memon N. Yeo B.-L. Craver, S. and M. M. Yeung. Resolving rightful ownerships with invisible watermarking techniques: limitations, attacks, and implications. *IEEE Journal on Selected Areas in Communications*, 1998.

[14] SPSS Inc. Training Department. *Advanced Statistical Analysis Using SPSS*. SPSS Inc., 2000.

[15] Stephen Drape and Clark Thomborson. Notations, 2006. Available from URL(http://www.cs.auckland.ac.nz/ stephendrape/papers/waternotation.pdf).

[16] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern classification*. John Wiley & Sons, 2nd edition, 2001.

[17] T. Furon and P Duhamel. An asymmetric watermarking method. *Signal Processing*, 2003.

[18] Roedy Green. How to write unmaintainable code. *Java Developers' Journal*, 2006.

[19] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, Inc, 1991.

[20] A. B. Kahng, J. Lach, W. H. Mangione-Smith, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe. Watermarking techniques for intellectual property protection. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 776–781, New York, NY, USA, 1998. ACM Press.

[21] T Kalker. Considerations on watermarking security. In *Multimedia Signal Processing*, 2001.

[22] D. Kirovski and F. Petitcolas. Blind pattern matching attack on watermarking systems, 2003.

[23] Matias Madou, Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *DRM '05: Proceedings of the 5th ACM workshop on Digital rights management*, pages 75–82, New York, NY, USA, 2005. ACM Press.

[24] Pierre Moulin and Joseph A. O'Sullivan. Information-theoretic analysis of information hiding. *IEEE Transactions on Information Theory*, 49(3):563–593, 2003.

[25] Ginger Myles and Christian Collberg. K-gram based software birthmarks. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 314–318, New York, NY, USA, 2005. ACM Press.

[26] Jasvir Nagra, Clark Thomborson, and Christian Collberg. Software watermarking: Protective terminology. In *Proceedings of the ACSC 2002*, 2002.

[27] Shari Lawrence Pfleeger Norman E. Fenton. *Software Metrics, A Rigorious & Practical Approach*. PWS Publishing Company, 2nd edition, 1997.

[28] M. J Norušis. *SPSS 12.0 statistical procedures companion*. Princeton Hall, 2003.

[29] Julie Pallant. *SPSS survival manual*. Allen and Unwin, 2nd edition, 2005.

[30] J. Palsberg, S. Krishnaswamy, K. Minseok, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking, 2000.

[31] N. Provos and P. Honeyman. Hide and seek: An introduction to steganography. *IEEE Security & Privacy*, 2003.

[32] A.K Raudys, S.J.; Jain. Small sample size effects in statistical pattern recognition: recommendations for practitioners. *Pattern Analysis and Machine Intelligence*, 1991.

[33] S. Raudys and V. Pikelis. On dimensionality, sample size, classification error, and complexity of classification algorithm in pattern recognition. *Pattern Analysis and Machine Intelligence*, 1980.

[34] Peter Sžor and Peter Ferrie. Hunting for metamorphic. Symantec Security Response. White Paper., June 2003.

[35] Diomidis Spinellis. Reading, writing, and code. *ACM Queue vol. 1, no. 7*, 2003.

[36] Clark Thomborson. Information-theoretic models of attacks on watermarks, July 2006. email communication.

[37] S. Voloshynovskiy, S. Pereira, and J.J.and Su J.K Pun, T.and Eggers. Attacks on digital watermarks: classification, estimation based attacks, and benchmarks. *Communications Magazine*, 2001.

[38] N. Morimoto W. Bender, D. Gruhl and A. Lu. Techniques for data hiding. *IBM Systems Journal*, 35, 1996.

[39] C Wang. *A security architecture for survivability mechanisms*. PhD thesis, University of Virginia, 2000.

[40] Andrew Webb. *Statistical pattern recognition*. John Wiley & Sons, 2nd edition, 2002.