Summer 2013

Secure design defects detection and correction

Wenquan Wang

Department:

SECURE DESIGN DEFECTS DETECTION AND CORRECTION

by

WENQUAN WANG

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2013

Approved by

Wei Jiang, Advisor
Dan Lin
Marouane Kessentini

ABSTRACT

Bad design and software defects often make source codes hard to understand and lead to maintenance difficulties. In order to detect and fix such defects, researchers have systematically investigated these issues and designed different effective algorithms to tackle the problems. However, most of these methods need source codes/models for defect detection and correction. Commercial companies, like banks, may not be willing to provide their source models due to data security. Therefore, it is a huge challenge to detect software detects by a consulting company as well as to keep source models confidential. This thesis analyze security issues in existing approaches related to defect detection and develop secure protocols to allow a software corporation and a consulting company to exchange data securely without revealing any private information, which make the approach practical in reality. The experimental results confirm the effectiveness of the proposed approach.

ACKNOWLEDGMENTS

I would like to express my gratitude to my academic advisor Dr. Wei Jiang for the remarks and engagement through the learning process of this master thesis. Without his funding supports and kindly help, it is impossible for me to finish all research work and graduate from Computer Science department. Furthermore I would like to thank the committee members, Dr. Dan Lin and Dr. Marouane Kessentini for their useful comments. Also, I like to thank my family members, who have supported me throughout entire process. Finally, I must thank all my dear friends who have helped me and encouraged me ever.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Page

LIST OF TABLES

Page

1. INTRODUCTION

1.1. PROBLEM BACKGROUND

In typical software life cycles, software maintenance mainly includes adding/removing functionalities, detecting maintainability defects, correcting them, and modifying the code to improve its quality. Although maintainability defects are sometimes unavoidable, they should be removed from the code base as early as possible. However, detecting and removing are difficult, time-consuming, and to some extent, a manual process. To detect design defects automatically, several automated detection techniques have been proposed [14] [13] [17] [23], which are proved effective to improve software quality. In these settings, detection rules are manually defined or based on a huge mount of quality metrics. However, for most small software companies, they do not have enough resources to design complicated detection tools and collect rich rules or quality metrics. Then it is worthy employing a consultant who has professional skills and experience to diagnose source models and correct potential unreasonable defects.

Unfortunately, even though a plenty of research work has been done regarding how to detect and remove software defects, few of them considered privacy issues in their approaches. Certainly, it is in commercial companies' best interests not to disclose source codes, source models or any private information to others in the process of software evaluation. Then it is a huge challenge for us to preserve privacy without sharing source models, quality metrics, detection rules and algorithms between them. To better understand the problem, this thesis will first review the typical process of software defect detection and correction, and then discuss the privacy issues in the process.

Software maintainability defects, also called design anomalies, refer to design situations that adversely affect the maintenance of software. Maintainability defects are unlikely to cause failures directly, but may cause them indirectly. In general, they make a system difficult to change, which may in turn introduce bugs. Software defect detection refers to find software defects with a set of quality metrics, and correction is the process to fix them with series of refactoring operations. For example, to correct the blob defect, many operations can be used to reduce the number of functionalities in a specific class: move methods, extract class, etc. Opdyke defines refactoring as the process of improving

a code after it has been written by changing its internal structure without changing its external behavior. The idea is to reorganize variables, classes and methods in order to facilitate future extensions. This reorganization is used to improve different aspects of software-quality: reusability, maintainability, complexity, etc.

The work [13] presents an effective approach to detect and correct software defects, which is based on Genetic Algorithm (GA). This approach mainly includes two steps. In the first step, it generates detection rules from an initial set of rules representing random combinations of metrics. Then, Genetic Programming (GP) is applied to refine this set progressively according to each individual rule's ability to detect defects in the example base. This process takes defect examples from source models as input, and its objective is to prevent sharing source models' information with consultants in this step. After generating the detection rules, this approach uses them in the correction step. It starts by generating some solutions that represent a combination of refactoring operations and then evaluate them by their ability to correct defects. Eventually, the best solution would correct most detected defects.

However, the approach [13] has some limitations, too. First, it has to access source models to execute detection rules; second, it will take source models as input to fix detected defects in correction step. All of these operations will disclose source models to consulting corporations so that it may not be acceptable by most commercial companies. In this thesis, new security protocols are proposed to overcome some of the mentioned limitations and the new approach will allow a consulting company to evaluate a commercial banks' software without revealing any private information. At the beginning, the concept of Trusted Third Party (TTP) [11] is introduced to model secure defect detection and correction. Two parties can communicate safely with the TTP and do not need to worry about security issues because TTP cannot disclose a party's private information to the other. Next, secure protocols are designed to replace the TTP such that during the execution of the protocols, the private information is never disclosed.

1.2. PROBLEM DEFINITION

To better understand the thesis' contribution, it is important to define the problems of defect detection and correction, and the privacy preserving process. This

section will first introduce the definitions of important concepts related to the new proposed protocols, and then emphasize on the specific problems that are tackled by the approach.

Defect Detection and Correction (DDC): The defect detection process consists of finding code fragments that violate structure or semantic properties on code elements such as coupling and complexity. The detected correction refers to fix these defects by applying refactoring operations.

The functions Defect Detection and Defect Correction could be defined as below.

$$\text{Defect\_Detection}(R_i, S) \rightarrow R_{i+1} \quad (1)$$

- Input: R_i represents the detection rules obtained in iteration i and S denotes the source model
- Output: R_{i+1} represents the detection rules generated and evaluated in iteration $i+1$

The algorithm generates initial detection rules R_0 from quality metrics and applies these rules to detect defects in source models. Then it will evaluate each rule set based on its ability to detect the number of defects. Next, it may generate new rule set R_1 and evaluate them again. Eventually, it outputs the best rule set $\hat{r}$, which can detect most defects in source models, after n iterations.

$$\text{Defect\_Correction}(\hat{r}, f_j, S) \rightarrow f_{j+1} \quad (2)$$

- Input: $\hat{r}$ denotes the best detection rules obtained from Defect Detection, and f_j represents refactoring operations generated in iteration j, S is the source model.
- Output: the refactoring set f_{j+1} generated and evaluated in iteration $j+1$

Initially, the algorithm generates a set of refactoring operations f_0 and applies them to fix detected defects in source models. Then it detects remaining defects with $\hat{r}$ and evaluates

the set of refactoring based on its ability to correct defects. Next, it will generate new refactoring operations f_1 and evaluate them too. At last, it will return the best refactoring set $\hat{f}$ which would fix most detected defects.

Secure Defect Detection and Correction (SDDC): preserve both parties' private information while following the general DDC process. Suppose P_1, e.g. a consulting company, owns a set of quality metrics and refactoring; P_2, e.g. a commercial bank, has private source models. SDDC allows P_1 to apply quality metrics to detect software defects in P_2's source models and fix them without revealing P_1's quality metrics, detection and correction algorithms to P_2; or disclosing P_2's source models to P_1. In detection stage, SDDC will take source models, quality metrics as input, and output best detection rules after iterations. Next, SDDC refines refactoring operations and evaluate them with these rules; eventually generates optimal refactoring solutions as output, which would correct most defects in source models.

Secure Defect Detection and Secure Defect Correction functions are defined as the following.

$$\text{Defect\_Detection}_s((P_1, R_i), (P_2, S)) \rightarrow (P_1, R_{i+1}) \quad (3)$$

- Input: (P_1, R_i) represents P_1's private detection rules obtained in iteration i and (P_2, S) is the source model of P_2
- Output: the rule set R_{i+1} which is generated and evaluated in iteration $i+1$, only P_1 gets R_{i+1}

Equation (3) is similar to Equation (2), and it starts from generating initial detection rule R_0. Eventually, it outputs the best rule set $\hat{r}$ which can detect most defects.

$$\text{Defect\_Correction}_s((P_1, \hat{r}, f_j), (P_2, S)) \rightarrow (P_1, f_{j+1}) \quad (4)$$

- Input: $(P_1, \hat{r}, f_j)$ includes two parts; $\hat{r}$ is the best rule set obtained from (3) which can detected most defects in source models, and f_j is P_1's refactoring generated in iteration j; (P_2, S) denotes the source model of P_2
- Output: the refactoring set f_{j+1} which is generated and evaluated by P_1 in iteration $j+1$

Similarly, Equation (4) starts from generating the initial set of refactoring operations f_0 to fix detected defects. In iteration j, it outputs f_{j+1} which will be the input of iteration $j+1$. At last, P_1 gets the best refactoring set $\hat{f}$ which would fix most detected defects and then P_1 will send $\hat{f}$ to P_2 to fix the defects in source models.

Equation (3) and Equation (4) are very similar to Equation (1) and Equation (2) except that they will preserve both parties' private information in the process of defect detection and correction. Both parties own some private items. For P_1, he has four private items: first, the quality metrics; second, the best rule set; third, the process of the best rule set generation; last, the process to generate the optimal refactoring set. Meanwhile, P_2 only owns private source models. The thesis will propose a new approach to implement algorithms defined by Equation (3) and Equation (4), and fulfill security property at the same time. The remainder of this thesis is organized as follows. Section 2 is dedicated to the related work and background. The TTP model is outlined in Section 3. In Section 4, the thesis gives an overview of secure protocols. Then, Section 5 discusses security and communication analysis and Section 6 presents the validation results. Future research directions are summarized and suggested in Section 7.

2. RELATED WORK

2.1. INTRODUCTION OF EXISTING APPROACHES

The techniques regarding detecting and fixing design defects range from fully automatic detection and correction to guided manual inspection. Design defect detection and correction can be classified into three broad categories: rules-based detection-correction, detection and correction combination, and visual-based detection.

In the first category, Marinescu [18] defined a list of rules relying on metrics to detect defects which are at method, class and subsystem levels. Erni and Lewerentz [6] introduce the concept of multi-metrics, n-tuples of metrics expressing a quality criterion (e.g., modularity) to evaluate frameworks and improve them. Both of the two existing solutions require users to manually define threshold values for metrics in the rules, which is the main limitation of them. To handle this problem, Alikacem and Sahraoui express defect detection as fuzzy rules, with fuzzy labels for metrics, e.g., small, medium, large, and evaluate the rules by means of membership functions. Although no crisp thresholds need to be defined, it is not obvious to determine the membership functions. Moha et al. [19], in their DÉCOR approach, they start by describing defect symptoms using an abstract rule language. These descriptions involve different notions, such as class roles and structures. The descriptions are later mapped to detection algorithms. In addition to the threshold problem, this approach uses heuristics to approximate some notions, which results in a high rate of false positives. Khomh et al. [15] extended DECOR to support uncertainty and to sort the defect candidates accordingly. The majority of existing approaches to automate refactoring activities are based on rules that can be expressed as assertions (invariants, pre- and post-conditions), or graph transformations. The use of invariants has been proposed to detect parts of program that require refactoring by Kataoka et al. [12]. Opdyke [22] suggested the use of pre- and post-condition with invariants to preserve the behavior of the software. All these conditions could be expressed in the form of rules. Heckel [10] considers refactorings activities as graph production rules (programs expressed as graphs). However, a full specification of refactorings would require a large number of rules. In addition, refactoring-rules sets have to be complete, consistent, non redundant, and correct. Furthermore, the algorithm

needs to find the best sequence of applying these refactoring rules. In such situations, search-based techniques represent a good alternative.

In the second category of work, these approaches refactor a system by detecting elements to change to improve the global quality. For example, in [21], defect detection is considered as an optimization problem. The authors use a combination of 12 metrics to measure the improvements achieved when sequences of simple refactorings are applied, such as moving methods between classes. The goal of the optimization is to determine the sequence that maximizes a function, which captures the variations of a set of metrics [9].The fact that the quality in terms of metrics is improved does not necessary means that the changes make sense. The link between defect and correction is not obvious, which make the inspection difficult for the maintainers.

The high rate of false positives generated by the automatic approaches encouraged other researchers to explore semiautomatic solutions. These solutions took the form of visualization-based environments. The primary goal is to take advantage of the human ability to integrate complex contextual information in the detection process. Kothari et al. [16] presented a pattern-based framework for developing tool support to detect software anomalies by representing potential defects with different colors. Later, Dhambri et al. [5] proposed a visualization-based approach to detect design anomalies by automatically detecting some symptoms. The visualization metaphor was chosen specifically to reduce the complexity of dealing with a large amount of data. Still, the visualization approach is not obvious when evaluating large-scale systems. Moreover, the visualized information is metric-based and is difficult to detect complex relationships. In Kessentini's approach [13], human intervention is needed only to provide defect examples. Finally, the use of visualization techniques is limited to the detection step.

2.2. KESSENTINI'S APPROACH

Kessentini investigated limitations of the existing approaches and proposed a search-based refactoring scheme, which is the most effective one now. In this section, first let's review the detection and correction phases of this algorithm, and then analyze its security issues in practice. Figure 2.1 shows the general structure of the approach. It includes two important steps: 1) defects detection and 2) correction. The detection step takes a base example (i.e., a set of defects examples) and a set of quality metrics as inputs, and generates a set of rules as output. The generation process can generate the best set of rules that detect the maximum number of defects.

The correction step takes the generated detection rules and a set of refactoring operations as inputs, and generates a sequence of refactoring as output. The process can generate the best set of refactoring that minimizes the number of detected defects using the detection rules.

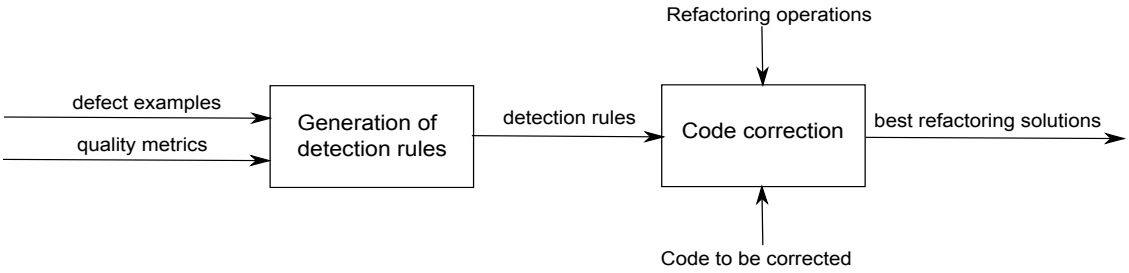

Figure 2.1. Overview of the approach

2.2.1. Defect Detection. The detection process starts from an initial set of rules representing random combinations of metrics. In order to understand the process, readers have to learn how to generate initial rules first. In fact, quality metrics (logic program) is represented as a forest of ANDOR trees. For example, consider the following logic program:

C1: defect (blob) :- locClass (upper, 1500), locMethod (upper,129).

C2: defect (blob) :- nmd (upper, 100).

C3: defect (spaghettiCode) :- locMethod (upper,151).

C4: defect (functionalDecomposition) :- nPrivField (upper,7), nmd (equal,16).

These logic programs can serve to build the defect detection rules. The set of rules C1-C4 can be described as the following:

R1 : IF (LOCCLASS $\geq$ 1500 $\wedge$ LOCMETHOD $\geq$ 129) $\vee$ (NMD $\geq$ 100) THEN defect = blob
R2 : IF (LOCMETHOD $\geq$ 151) THEN defect = spaghetti code
R3 : IF (NPRIVFIELD $\geq$ 7 $\wedge$ NMD = 16) THEN defect = functional decomposition

Thus, the first rule is represented as a sub-tree of nodes (AND-OR, metrics) as shown in Figure 2.2. The main program tree will be a composition of three sub-trees: R1 AND R2 AND R3. This example contains several special terms whose meanings are listed as below and shown in Table 2.1.

Blob: It is found in designs where one large class monopolizes the behavior of a system (or part of it), and other classes primarily encapsulate data.
Spaghetti Code: It is a code with a complex and tangled control structure.
Functional Decomposition: It occurs when a class is designed with the intent of performing a single function. This is found in code produced by non-experienced object-oriented developers.

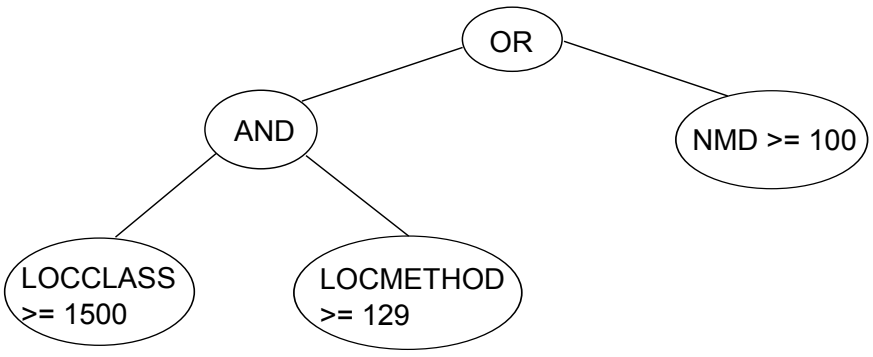

Figure 2.2. A tree representation of an individual rule

Table 2.1. List of related notation

Notation	Description
LOCCLASS	the number of lines of code in each class
LOCMETHOD	the number of lines of code in each method
NMD	the number of methods
NPRIVFIELD	the number of private fields

After initial rule set generation, this set is refined progressively according to its ability to detect defects present in the example base. Due to the very large number of possible rules (metric combinations), it uses a rule induction heuristic, called Genetic Programming (GP) to find a near-optimal set of detection rules. This approach's defect detection algorithm is given in Algorithm 1.

In fact, Equation (1) describes only an iteration of GP, but Algorithm 1 shows the whole process. It takes initial rule set and source models containing defect examples as input. Lines 2 construct an initial GP population, based on a given rule set R_0. The population stands for a set of possible solutions representing detection rules (metrics combination). Lines 4-20 encode the main GP loop, which searches for the best metrics combination. During each iteration, it evaluates the quality of each solution (individual) in the population, and the solution having the best fitness is saved. It generates a new population of solutions using the crossover operator (line 18) to the selected solutions; each pair of parent solutions produces two children (new solutions). It includes the parent and child variants in the population and then applies the mutation operator to each variant; this produces the population for the next generation. The algorithm terminates when it achieves the termination criteria (maximum iteration number), and return the best set of detection rules (solution).

Algorithm 1 Defect_Detection(R_0,S)

Require: R_0:initial rule set, S: source models with defects examples.

1: i=0

2: *initial_population*=R_0

3: $fitness\_\hat{r}=0$

4: **while** $i \leq m$ **do**

5: $fitness\_\hat{r}_i=0$

6: **for all** r_j in R_i **do**

7: $detected\_defects\_r_j$=Execute_Rules(r_j,S)

8: $fitness\_r_j$=Compare($detected\_defects\_r_j$,S)

9: **if** $fitness\_\hat{r}_i < fitness\_r_j$ **then**

10: $fitness\_\hat{r}_i=fitness\_r_j$

11: $\hat{r}_i=r_j$

12: **end if**

13: **end for**

14: **if** $fitness\_\hat{r} < fitness\_\hat{r}_i$ **then**

15: $fitness\_\hat{r} = fitness\_\hat{r}_i$

16: $\hat{r}=\hat{r}_i$

17: **end if**

18: R_{i+1}=Generate_New_Population(R_i)

19: $i=i+1$

20: **end while**

21: return $\hat{r}$

GP is introduced here to generate new rules. It generates new offsprings using selection, crossover or mutation in each iteration. New generated rules will be executed in next iteration and it will be saved as the new best solution if its fitness value is greater than current saved rules.

- **Selection**

  For the initial prototype, it uses stochastic universal samplying (SUS) selection algorithm, in which each individual's probability of selection is directly proportional to its relative fitness in the population.

- **Mutation**

It starts by randomly selected a node in the tree. Then, if the selected node is a terminal (quality metric), it is replaced by another terminal (metric or another threshold value); if it is a function (and-or), it is replaced by a new function; and if tree mutation is to be carried out, the node and its sub-tree are replaced by a new randomly generated sub-tree. Figure 2.3 shows an example of the mutation operation.

- **Crossover**

Two parent individuals are selected and a sub-tree is picked on each one. Then crossover swaps the nodes and their relative sub-trees from one parent to the other. Figure 2.4 shows an example of the crossover process. The rule R_1 and a rule R_2 form another individual (solution) are combined to generate new two rules.

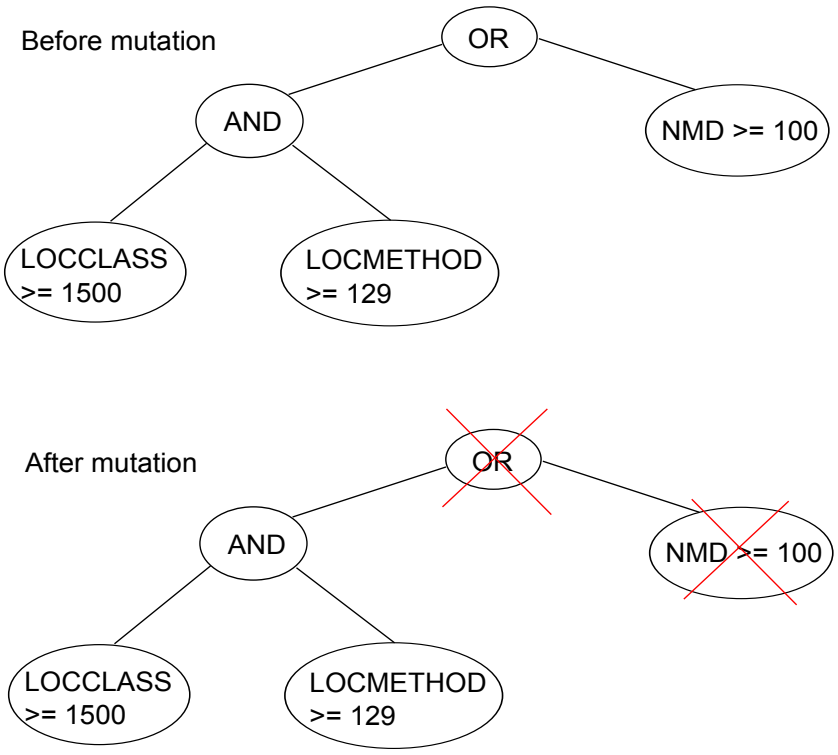

Figure 2.3. Mutation operator

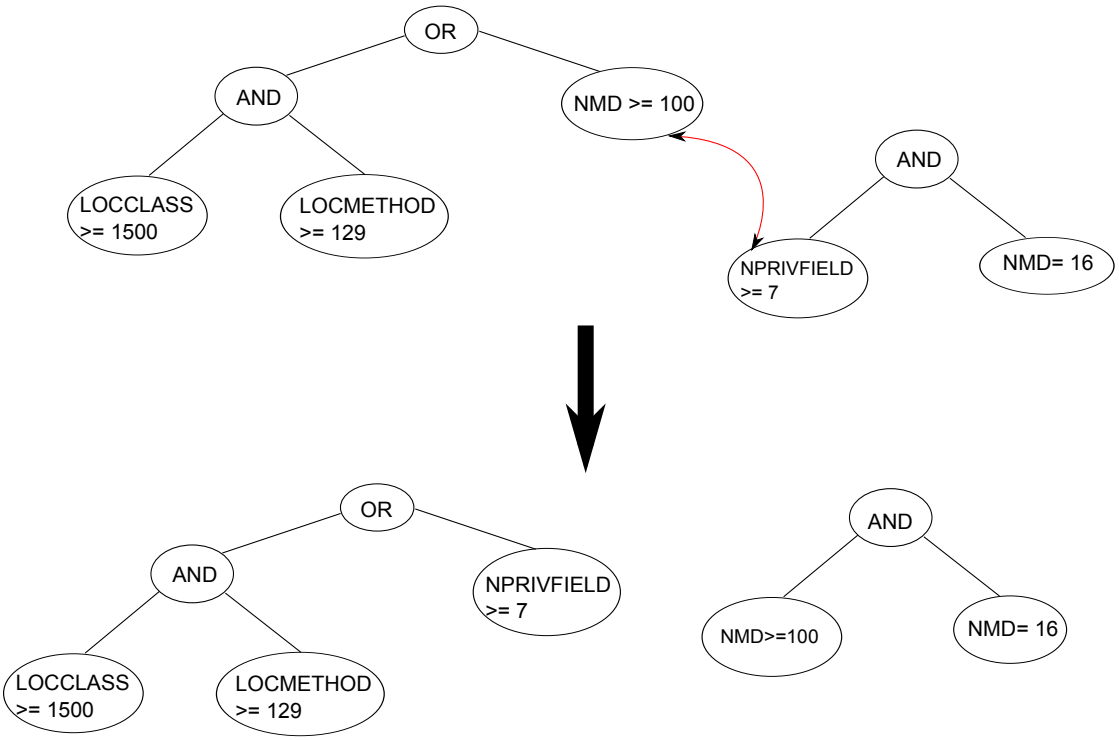

Figure 2.4. Crossover operator

2.2.2. Defect Correction After generating the detection rules, it uses them in the correction step. As shown in Algorithm 2, it starts by generating the initial solution f_0 that represents a combination of refactoring operations to apply. The defect correction algorithm takes the best detection rule set $\hat{r}$, initial refactoring set f_0 and source models as input. Then it executes the refactoring sequence on source models. Next, a fitness function calculates, after applying the proposed refactoring, the number of remaining defects using the detection rules. At last, the best solution ^ f which has the minimum fitness value is returned. Due to the large number of refactoring combination, a Genetic Algorithm (GA) is used.

Algorithm 2 Defect_Correction($\hat{r}$,f_0,S)

Require: $\hat{r}$:the best rule set, f_0:initial refactoring operations, S: source models.

1: *initial_population*=f_0

2: i=0

3: *fitness_* $\hat{f}$ =MAX_INTEGER

4: **while** $i \leq n$ **do**

5: Excute_Refactorings(f_i, *S*)

6: *detected_defects*=Execute_Rules($\hat{r}$,*S)*

7: *fitness_*f_i= | *detected_defects* |

8: **if** *fitness_* $\hat{f}$ **>** *fitness_*f_i **then**

9: *fitness_* $\hat{f}$ =*fitness_*f_i

10: $\hat{f} = f_i$

11: **end if**

12: f_{i+1} = Generate_New_Population(f_i)

13: *i*=*i*+1

14: **end while**

15: return $\hat{f}$

The approach views the set of potential solutions as points in an n-dimensional space, where each dimension corresponds to one refactoring operation, or called logic predicate. Initially, it generates a sequence of refactoring and executes them on the detected defects. Then, Genetic Algorithm is applied. The crossover operator creates two offspring from the two selected parents and the mutation operator will randomly change a dimension (refactoring) with a new refactoring. After applying crossover and mutation operators, the algorithm will generate a set of new refactoring. Then the new refactoring operations will be executed on source codes again.

Every set of generated refactoring can be viewed as a new correction solution and a defined fitness function quantifies the quality of the proposed refactoring. In fact, the fitness function checks to minimize the number of detected defects using the detection rules. At last, the algorithm will generate the best correction solutions, which are combinations of refactoring operations, and should minimize, as much as possible, the number of defects detected using the detection rules.

3. SECURITY ISSUES AND A NAIVE SOLUTION BASED ON TTP

3.1. SECURITY ISSUES IN KESSENTINI'S APPROACH

In this subsection, the thesis will analyze security issues in Kessentini's scheme and then propose a naive solution based on TTP. As said in Section 1.1, in the whole process, P_1 has four private items and P_2 owns private source models. In fact, these private items could be classified as two different types of privacy: data privacy and algorithm privacy.

- **Data Privacy**

  Certainly, quality metrics and the detection rules which are combinations of these metrics are valuable for P_1. Meanwhile, for P_2, source models are its private data. Therefore, both parties' data privacies in the whole process have to be preserved and each party's private data should not be disclosed to the other. In order to preserve data privacy, such secure protocols are desired, which implement all features of Equation (1) and (2), and also have security property at the same time. In fact, most interaction and data exchange happen in the two functions Execute Rules and Compare in detection and correction algorithms, so the main goal of this thesis is to design secure versions of the two functions.

- **Algorithm Privacy**

  Besides private data, the processes of finding best detection rule set and best refactoring operations are private, too. P_1 will not allow P_2 to learn them or apply these algorithms to evaluate its software by itself later. Therefore, the secure protocols should fulfill data privacy and preserve algorithm privacy, too.

3.2. INTRODUCTION OF TTP

The goal is to implement Equations (3) and (4) with security property and the thesis will take two steps to achieve such target. First, it redesigns Equations (1)/(2) to fulfill the requirements Equations (3)/(4) by adding a trusted third party in the process. Then, it designs new secure protocols that can act as the same roles as TTPs. A trusted third party (TTP) can be described as an entity trusted by other entities with respect to

security-related services and activities. TTP is an impartial intermediary whose role is to ensure that each party receives the item it expects. It is assumed that the TTP is neutral, available and trusted by all groups. Sometimes, more than one TTP might be involved in a transaction. Typically, a TTP will be an organization licensed or accredited by a regulatory authority, which will provide security services, on a commercial basis, to a wide range of bodies, including those within the telecommunications, finance and retail sectors.

The use of TTPs is dependent on the fundamental requirement that the TTP is trusted by the entities it serves to perform certain functions. In practice, TTPs could exist in both public and corporate domains, at the local, national and international level. TTPs should have trust agreements arranged with other TTPs to form a network, thus allowing a user to communicate securely with every user of every TTP with whom his TTP has an agreement. Any TTP scheme should also allow for both national and international operation, allowing users in any country, where an appropriate TTP resides, to communicate securely. TTPs can be categorized according to their communication relationships with the users they serve [5], [6]. A TTP may provide its services through a combination of the different modes for different parts of its service.

- **Off-line TTPs**

  An off-line TTP does not interact with the user entities during the process of the given security service unless a problem occurs. Fox example, the two parties directly trade their items, and in case of any problem, the TTP will be involved to mediate between the parties.

- **On-line TTPs**

  An on-line TTP is requested by one or both entities in real-time to provide, or register, security-related information. Such a TTP is not in the communications path between the two entities; rather, it is for verifying an item, and generating and/or storing proof of exchange of items.

- **In-line TTPs**

  An in-line TTP is positioned in the communication path between the entities. Such an arrangement allows the TTP to offer a wide range of security services directly to users. This means that the TTP receives the items from each party,

authenticates them and delivers them to the respective parties. Since the TTP interrupts the communication path, different security domains can exist on either side of it.

3.3. A NAIVE SOLUTION BASED ON TTP

Because P_1 and P_2 cannot share and exchange their private items directly, it is reasonable to design an in-line TTP model in this scenario in order to preserve data and algorithm privacies.

In detection stage, data privacy includes P_1's quality metrics and detection rules; P_2's source models. To preserve data privacy in this step, P_1 and P_2 should send detection rules and source models to an in-line TTP, respectively. Then, TTP will execute these rules on source models and compute the rule set's fitness score. After that he sends the fitness back to P_1 who then updates its best rule set if the received fitness score is greater than current one. In addition, to preserve algorithm privacy, P_2 should not learn the GP iteration process, so it is better to request P_1 to apply GP to generate new rules and sends them to TTP for execution, and TTP returns calculated fitness score to P_1 for evaluation. The iteration process continues and finally, P_1 will find the best detection rules. In this model only TTP knows both parties' private data and algorithms; P_1 and P_2 will learn nothing regarding the other's private information. Figure 3.1 shows the process and the role of TTP in detection.

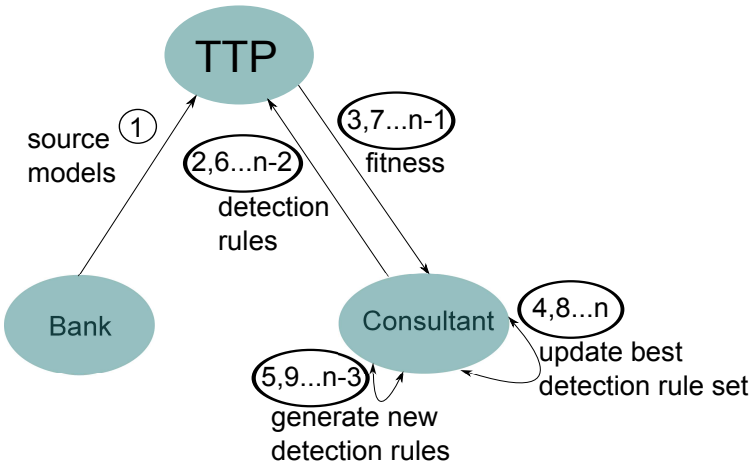

Figure 3.1. TTP model in defect detection

In correction step, because P_1 should call the routine Execute Rules and Compare to detect remaining defects in each correction iteration, data privacy is the same as that of detection phase. Only difference is algorithm privacy, and so the secure protocols have to keep the original process to generate refactoring solutions private. Therefore, the responsibilities of TTP role in this step are to preserve the same data privacy as that of detection and keep the refactoring generating process safe. In each iteration, P_1 sends refactoring operations to TTP who will execute them on source models. Next, TTP applies detection rule set to detect remaining defects and inform P_1 the refactorings' fitness. P_1 saves the refactoring set as current best solution if it has a smaller fitness score. Then P_1 will generate new refactorings and require TTP to evaluate them. At last, P_1 obtains the best solutions and then send them to P_2 to fix most detected defects. Figure 3.2 shows the correction step and its output.

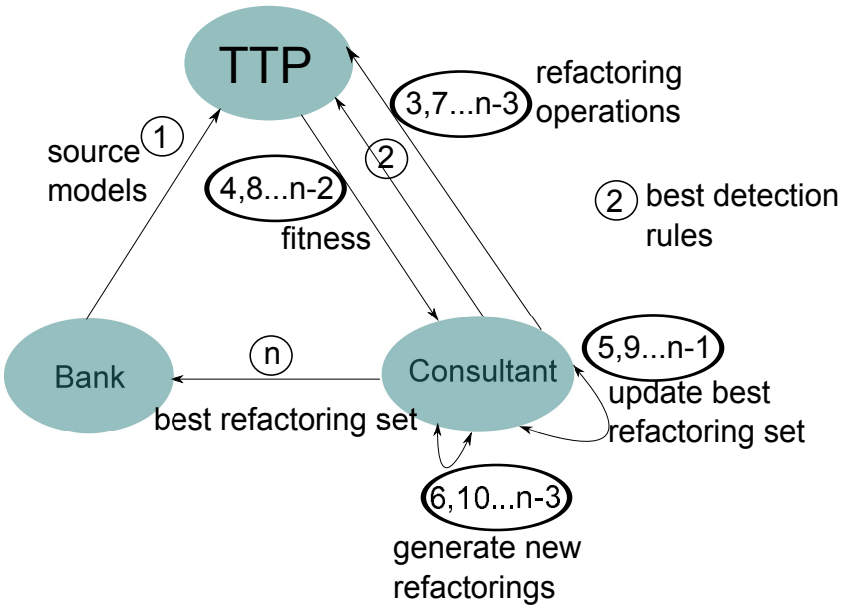

Figure 3.2. TTP model in defect correction

3.4. COMPARISON TO IDEAL TTP MODELS

The proposed TTP models are designed to follow the process of Genetic Algorithm, so P_1 has to interact with the TTP role for many rounds. However, they are

not ideal TTP models because the communication rounds between a TTP role and P_1 would leak the fitness score of a rule set to P_1. In an ideal model, P_1 and P_2 send quality metrics and Genetic Algorithm; source models to a TTP, respectively. Next, the TTP runs A to find best detection rules and optimal refactoring solutions. Finally he applies the solutions to fix existing defects in source model and then returns it to P_2. In the process, the TTP do not interact with P_1, so no extra information is disclosed.

4. PRIVACY-PERSERVING DDC PROTOCOLS

4.1. THE ROLE OF SECURE PROTOCOLS

TTP is an ideal model, but in reality it is hard to find a fully trusted third party. Even through, TTP model is definitely the guide for secure protocol design. If a new protocol is proved to be able to replace entire TTP role in the model, then the protocol is secure and implements all functions of TTP role.

The data and algorithm privacies of Kessentini's approach are analyzed in Section 3.1. Algorithm privacy is not hard to preserve because it is straightforward to require P_1 to execute most steps in the process and he only interacts with P_2 when he has to do that. However, to preserve data privacy is not easy because the approach executes generated rules on source models to obtain the best rule set. How to keep the entire private data secret during the execution? In fact, as mentioned in Section 3.1, to preserve privacy, it is indispensable to design secure version for function Execute Rules and Compare. In TTP model, all private data is sent to TTP and the two routines are executed by TTP too. Now, it is very possible to design secure protocols to replace the TTP role.

First, let us analyze the routine Execute Rules. In every iteration of detection phase, each new generated rule is a combination of quality metrics. In order to apply these rules, the function Execute Rules will compare each rule's thresholds with source models' information, e.g. LOCCLASS, NMD, to determine if software defects exist in a class, which means P_1 only concerns whether these statistical indicators of each class are greater or less than thresholds of its detection rules instead of their actual values. Based on such investigation, secure comparison techniques and secure multi-party computation (SMC) can be applied to perform such comparison. P_1 will execute his rules and evaluate P_2's software quality according to secure comparison results without learning any private information of source models. Also, P_2 cannot learn P_1's quality metrics, detection rules from the secure comparison protocol.

Now, a plenty of research work has been done regarding secure comparison. General two-party computation was introduced by Yao [29], and general computation for multiple parties was introduced in [3]. Most of the existing secure protocols focus on

solutions of secure integer comparison problem and their applications, e.g. online auction, data mining without learning more details [2], association rule mining [24], web services [1], etc. Secure integer comparison (SC) is the starting point of SMC protocols. There are a plenty of specialized solutions to the problem which provides efficiency with respect to generic methods [7] . Most of these solutions are based on doing calculations on the bits of integers by using homomorphic encryption or encrypting bits as quadratic residues and non-residues modulo an RSA modulus. The work [20] shows that it is more efficient than previous ones. Therefore, the thesis integrate [20] to the designs and apply it to handle secure issues in rule execution.

Second, the thesis discusses how to apply secure protocols to preserve privacy in the function Compare. Actually, the routine Compare is to compute fitness score for each rule set and in a word it is to calculate what percent of true defects are found by the detection rules. The secure protocols should compute the fitness without disclosing P_2's true defects to P_1. To compute the fitness score, it is the key point to get the number of detected true defects. In fact, it is not hard to imagine that base examples contain true defect set defined manually by experienced engineers and detected defects are included by another set, then the problem to find the number of detected true defects can be transformed to compute the intersection of two sets. Dot product for set intersection computation is another category of secure protocols and it is a perfect solution to tackle the security issues in the routine Compare. In the following section, the thesis will discuss the details of how to apply this technique to design secure version of the Compare function.

4.2. SECURE INTEGER COMPARISON

Secure multi-party computation (SMC) was first suggested by Yao[1] as the millionaires problem, in which two millionaires want to learn who is richer without revealing their wealth to each other. The problem with its solution gave rise to the more general problem, where multiple parties try to compute some function securely given each party contributes some secret input. Several secure integer comparison (SC) protocols [8] [3] [7] have been widely studied and proposed. Recently, the work [20] proposed a new secure comparison protocol that can be applied to check some integer

over an interval securely. It uses a perfect binary tree (PBT), in which the leaf level contains all possible integers, 0 through n1, and this protocol is designed to compare two integers at leaf level. Properly speaking, the secure integer comparison scheme with arguments (a, b) is a two-party protocol between P_1 and P_2 who have n bit inputs a and b respectively. At the end of the protocol, P_1 learns if b _ a without learning b.

This scenario is exactly the same as the situation in threshold evaluation, and then it is possible to apply SC to get the comparison results. To understand this scheme, first, the concept of PBT and some definitions will be covered. In a word, a PBT is a full binary tree and all non-leaf nodes exactly have two children. Here a unique label (h, o) is used to represent a node in PBT, where h denotes the node's height and o denotes its order in the layer.

Before readers start to understand this algorithm, some special terms should be learnt first.

Coverage: Given a PBT, it is said that a tree node (h1, o1) covers a leaf node (0, o2) if there exists a path from (h1, o1) to (0, o2) in the tree. The covering set of a given leaf node v is the set of all nodes in the PBT that cover v. The coverage of a tree node v is the set of all leaf nodes covered by v. For example, in Figure 4.1, (2, 1) covers (0, 6). Covering set of the leaf node (0, 6) is {(0, 6), (1, 3), (2, 1), (3, 0)}. The coverage of (2, 1) is {(0, 4), (0,5), (0, 6), (0, 7)}.

Representer Set: is a minimal set that is the coverage of all leaves in a set of leaf nodes. In Figure 4.1, {(1, 1)} is a minimal representer for {(0, 2), (0, 3)}, and {(0, 4)} is a minimal representer for {(0, 4)}. Then {(1, 1), (0, 4)} is a minimal representer for {(0, 2), (0, 3), (0, 4)}.

Homomorphic Encryption [4]: is a form of encryption which allows specific types of computations to be carried out on ciphertext and obtain an encrypted result. For some prime p, it has the following properties.

$$E(m_0) \cdot E(m_1) = E(m_0 + m_1)$$

$$E(m)^c = E(c \cdot m)$$

In the algorithm Secure Comparison [20], P_1 wants to compare its private integer a to P_2's private b. First, P_1 creates a representer set for the leaf nodes (0, 0) … (0, a). For each level *i* in the PBT, P_1 creates a polynomial T_i whose root is the order of the representer node with height *i*. P_1 uses an additively homomorphic public key encryption scheme, E, to encrypt the coefficients and sends the encrypted polynomials to P_2 who calculates the covering set B of the node (0, b). For each node *v* in B, he securely evaluates polynomial $P_{v \cdot h}$ on $v \cdot o$ with help of the homomorphic property of the encryption. He multiplies the results with positive random numbers, and sends the shuffled results back to P_1 who will learn $b \leq a$ if any of the results decrypts to 0.

As an example of the algorithm, suppose P_1 holds $a = 5$, P_2 holds $b = 2$. Then, P_1 creates the representer {(1, 2), (2, 0)} for the set of leaf nodes {(0, 0)…(0, 5)} which represents *a*. Next, P_1 generates cofficient set {-1, -2, 0}. He sends encrypted coefficients $E_{pk}(1)$, $E_{pk}(-2)$, $E_{pk}(0)$ to P_2 in order. P_2 finds the covering of (0,2); {(0,2),(1,1),(2,0)} and calculates $(E_{pk}(2) * E_{pk}(1))^r * E_{pk}(0)$, $(E_{pk}(1) * E_{pk}(-2))^r * E_{pk}(0)$, $(E_{pk}(0) * E_{pk}(0))^r * E_{pk}(0)$ and sends back to P_1 in random order. P_1 sees one of the outputs decrypts to 0, she concludes $b \leq a$. It is not hard to explain the theory of secure integer comparison in a simple sentence. Because P_1's representer covers all leaf nodes less or equal to *a*, then *b*'s coverage must include one of nodes in the representer if $b \leq a$.

In detection step, the secure protocols may apply the protocol as below. P_1 creates the representer for a threshold and compute coefficient set. Then he sends encrypted coefficients to P_2 in order. P_2 finds the coverage of its corresponding statistical indicator and calculates the product of encrypted coverage and coefficients. Finally he sends them back to P_1 randomly and P_1 will learn which one is greater based on the decryption output.

The security of detection algorithm is based on the inability of either side to learn the other side's item without private key. In this protocol, the only way to decrypt P_1's encrypted representer is to learn the private key, unfortunately P_2 cannot learn the key

because it belongs to P_1. P_2 knows the public key and then he can encrypt its data with a random *r*. Similarly, P_1 received $EPR[i] = (EP[i] * E(B[i].o))^r * E_{pk}(0) = E[r *(P[i] + B[i].o)]$, then he cannot learn P_2's original data for he doesn't know the random number *r* and data's exact order. He only learns whether the sum of two integers is zero or not based on the property of homomorphic encryption.

4.3. THRESHOLD EVALUATION ALGORITHM

Once a secure comparison solution is found, it is not hard to integrate it to the threshold evaluation algorithm which executes each rule securely by calling the secure comparison routine. As shown in Algorithm 3, an individual rule can be divided into threshold set and operator set. First, it calls Secure Comparison to compare each pair of integers; threshold and corresponding information of source models. Then, apply operators to the comparison set to get variable b which is either 1 if the class contains a defect or 0, otherwise.

Algorithm 3 Threshold_Evaluation$_s$((P_1, *T*, *O*), (P_2, *S*))

Require: P_1: *T*(Threshold set) = {t_1, t_2,. . . ,t_m}, *O*(Operation set) ={o_1, o_2,. . . ,o_{m-1}},
P_2: *S*(Statistics information of source models) = { s_1, s_2,…, s_m}
1: *C*(Comparison set) = { c_1, c_2,…, c_m}, where c_i = Secure_Comparison(s_i, t_i)
2: $b = c_1\ o_1\ c_2\ o_2 \ldots o_{m-1}\ c_m$
3: return *b*

Take rule R1 as an example, for rule R1, T={1500, 129, 100}, O={∧, ∨}, S= {LOCCLASS, LOCMETHOD, NMD}. Then C={LOCCLASS ≥ 1500, LOCMETHOD ≥129, NMD ≥100} and R={*c1* ∧ *c2* ∨ *c3*}. Thus, if *b* is 1, then the detected class is a blob, otherwise if *b* is 0, no blob defect in this class.

4.4. USING SECURE SET INTERSECTION TO COMPUTE FITNESS

As data privacy mentioned in Section 3, the main objective is to implement secure protocols for Execute Rules and Compare functions. Now a secure comparison algorithm

is proposed, which can replace Execute Rules and allow two parties to evaluate each individual rule securely. In this subsection, the thesis will discuss how to apply secure set intersection techniques to compute fitness and replace the function Compare.

First, let's learn how to compute an individual rule set's fitness score. The fitness function checks to maximize the number of detected defects in comparison to the expectedones in the base of examples. Kessentini's approach [13] defined the fitness function as

$$f = \frac{\frac{\sum_{i=1}^{p} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}}{2}$$

In the function, f is normalized in the range [0, 1]; p is the number of detected classes and t is the number of defects in the base of example; a_i has value 1 if the ith detected classes exists in the base example (with the same defect type), and value 0 otherwise. From the function, it is clear that the summation of a_i is actually the size of intersection between detected defects and defects in the example base. In fact, P_2 may not be willing to disclose true defects in the base example to P_1 because P_1 might create fake rules conformed to these true defects to show false effectiveness of his solution, otherwise. Thus, it is better to keep the true defects private while computing fitness. A secure Compare function is already proposed, by which P_2 can obtain the size of intersection of two defect set and thus learn the effectiveness of this rule set. Once P_2 gets the size of intersection set, it is straightforward to calculate the fitness by applying the proposed fitness function.

Algorithm 4 $\text{Compare}_s((P_1, D), (P_2, E))$

Require: P_1: DS(Detected defect set) = $\{ d_1, d_2, \ldots, d_m \}$; P_2: ES(Defect examples in source models) = $\{ e_1, e_2, \ldots, e_m \}$; E and D are additively homomorphic semantically secure encryption/decryption functions, respectively; p_k is the public key.

1: P_2: EE(Encrypted defect examples) =$\{ \quad ee_1, ee_2, \ldots, ee_m \}$, where $ee_i = E_{pk}(e_i)$

2: P_2: send EE to P_1

3: P_1: P(Product set) = $\{ p_1, p_2, \ldots, p_m \}$, where $p_i = d_i \times ee_i$

4: $e = 1$

5: **for all** $p_i \in P$ **do**

6: **if** $p_i \neq 0$ **then**

7: P_1: $e = e \times p_i$

8: **end if**

9: end for

10: P_1: sends e to P_2

11: P_2: $d = \mathrm{D}(e)$

12: P_2: learns the effectiveness of this rule set

13: P_2: computes the fitness and return it to P_1

In Algorithm 4, all elements in the input sets *DS* and *ES* are binary numbers, whose values are either 1 or 0. First P_2 uses homomorphic encryption algorithm to encrypt true defects and then sends the sequence to P_1 in order, who will compute each p_i. Afterward the product of all non-zero p_i is calculated and its decryption result shows the size of *DS* and *ES*'s intersection. Next, P_2 computes the fitness score of this rule set and return it to P_1, who will update his optimal rule set based on this score.

4.5. SECURE PROTOCOL FOR DEFECT DETECTION

As mentioned in section 2, one of the research goals is to implement Defect_Detection$_s$. Now it is already described that how to apply SC to preserve privacy in routine Execute Rules; how to compute fitness securely by set intersection algorithm. Thereby, it is not hard to design secure protocols for detect detection. As the protocol shown in Algorithm 5, the thesis divide the original process into two sequences of actions performed by P_1 and P_2, respectively. They call Threshold Evaluations and Compares to preserve data privacy, and P_1 controls the process of GP to achieve algorithm privacy.

Algorithm 5 Defect_Detection$_s$((P_1, R_0), (P_2, S, E))

Require: (P_1, R_0): P_1's initial rules, (P_2, S, E): P_2's source models and defect examples.

1: P_1: $i = 0$
2: P_1: $initial\_population = R_0$
3: P_1: $fitness\_\hat{r} = 0$
4: **while** $i \leq m$ **do**
5: P_1: $fitness\_\hat{r}_i = 0$
6: **for all** r_j in R_i **do**
7: P_1: $detected\_defects\_r_j = 0$
8: **for all** $class_k$ in S **do**
9: P_1, P_2: (P_1, b) = Threshold_Evaluation$_s$((P_1, T_{rj} , O_{rj}), (P_2, $class_k$))
10: **if** $b = 1$ **then**
11: P_1: $detected\_defects\_r_j = detected\_defects\_r_j + 1$
12: **end if**
13: **end for**
14: P_1: $fitness\_r_j$ = Compare$_s$((P_1, $detected\_defects\_r_j$), (P_2, E))
15: **if** $fitness\_\hat{r}_i < fitness\_r_j$ **then**
16: P_1: $fitness\_\hat{r}_i = fitness\_r_j$
17: P_1: $\hat{r}_i = r_j$
18: **end if**
19: **end for**
20: **if** $fitness\_\hat{r} <$ fitness$_\hat{r}$ i **then**
21: P_1: $fitness\_\hat{r} = fitness\_\hat{r}_i$
22: P_1: $\hat{r} = \hat{r}_i$
23: **end if**
24: P_1: R_{i+1} = Generate_New_Population(R_i)
25: P_1: $i = i + 1$
26: **end while**
27: return $\hat{r}$

4.6. SECURE PROTOCOL FOR DEFECT CORRECTION

Once P_1 finds the best detection rules, he will choose proper refactoring to fix all detected defects. For correction, they also need to exchange information to correct existing defects and evaluate the effectiveness of refactoring operations. In the process, P_1 chooses a refactoring set for the current defects and sends them to P_2 for execution. P_2 will run the refactoring operators on source models and then they will exchange information to compute the fitness for this refactoring sequence. Next, P_1 may apply Genetic Algorithm to generate new offsprings or new refactorings and follow the same procedure as previous to evaluate them. The iteration continues and finally, the process outputs the best refactoring set which can fix most defects. Algorithm 6 shows the secure correction process.

Algorithm 6 Defect_Correction$_s$((P_1, $\hat{r}$, f_0), (P_2, S))

Require: (P_1, $\hat{r}$, f_0): $\hat{r}$ is the best rule set, f_0 is initial refactoring operations; (P_2, S): S is the source model.

1: P_1: *initial_population* = f_0

2: P_1: i = 0

3: P_1: *fitness_* $\hat{f}$ = MAX_INTEGER

4: **while** $i \leq n$ **do**

5: P_2: Execute_Refactorings(f_i, S)

6: P_1: *detected_defects* = 0

7: **for all** *class*$_k$ in S **do**

8: P_1, P_2: (P_1, b) = Threshold_Evaluation((P_1, $T\_\ \hat{r}$, $O\_\ \hat{r}$), (P_2, class$_k$))

9: **if** b = 1 **then**

10: P_1: *detected_defects* = *detected _ defects* + 1

11: **end if**

12: **end for**

13: P_1: *fitness_*f_i = |*detected_defects*|

14: **if** *fitness_* $\hat{f}$ > *fitness_*f_i **then**

15: P_1: *fitness* _ $\hat{f}$ = *fitness*_f_i

16: P_1: $\hat{f} = f_i$

17: **end if**

18: P_1: f_{i+1} := Generate_New_Population(f_i)

19: P_1: $i = i + 1$

20: **end while**

21: return $\hat{f}$

4.7. COMPLEXITY ANALYSIS

The total running cost depends on the number of candidate item sets , e.g. number of detection rules and thresholds, number of refactorings, the rounds of GA and GP. Suppose in detection step *k* original rules are generated from quality metrics and source models contain l classes; each iteration will generate m new rules and the GP iteration will terminate after n rounds, then time complexity is *O(k*l+m*n*l).* Similarly, in correction step the time complexity depends on initial refactorings, the number of generated new refactorings in each iteration and iteration rounds.

In this scheme, the running cost is highly related to comparison times because the algorithm would encrypt data in each comparison round, which is the most time-consuming action in the comparison process. In addition, SC protocol should be called for each threshold of every individual rule, so the number of total execution rounds is inevitable huge. Suppose each rule has r average thresholds, then SC would be executed *r*(k*l+m*n*l)* times. The thesis will verify the performance of SC protocols and discuss how to improve it in experimental results section.

4.8. COMPARISON TO TTP MODELS

Secure protocols are already implemented to preserve data and algorithm privacy and they can replace TTPs to some extent. Now let us compare the two types of different secure solutions and analyze what information is disclosed in the process. In the detection and correction TTP models, P_1 and P_2 never interact except that finally P_1 sends

refactoring solutions to P_2. TTPs execute each individual rule, every refactoring operation and compute fitness, so no private information would be revealed. However, in secure protocols, they have to communicate to execute rules and compute fitness, then P_1 or P_2 may learn something in the communication rounds. For example, P_1 would generate a plenty of rules and a certain number of them may evaluate a same index, e.g., a rule contains 'IF (NMD $\leq$ 100)'; another rule includes 'IF (NMD $\geq$ 90)', if both comparison results are true, P_1 would know the interval of NMD and even the exact value in some cases.

Moreover, P_2 will learn how to compute fitness while he calls the routine Compares, which is the information disclosed in this protocol. By contrast, if P_1 is requested to calculate the fitness, it should know the total number of true defects which is an input parameter of the fitness function. In short, some information has to be revealed by the protocol Compares anyway.

However, compared to TTP models, the proposed secure protocols preserved most private data and algorithm information even if there exist risks to leak minor part of them. For example, in the routine Execute Rules, P_1's private quality metrics and detection rules are kept secret; P_2's source models are never disclosed to P_1 too. In addition, Compare function keeps true defect information private and P_1 cannot learn it.

5. A SIMPLE EXAMPLE OF THE PROPOSED PROTOCOLS

This section will introduce an example case here to review the secure defect detection and correction process. In the process, rules R1, R2 and R3 are used to detect source models. Take a piece of source models in appendix as example and suppose class PrjInfos contains more than methods and the class is over 1500 lines, then it is a blob based on rule R1. Similarly, suppose class GanttApplet contains spaghetti code and class DeprecatedProjectExport- Data violates rule R3.

In detection step, P_1 will generate initial rule set R1, R2, R3 and request P_2 to collect related information from source models for comparison. For instance, R1 requires LOCCLASS and LOCMETHOD, then P_2 should count the number of code lines in each class and the number of code lines of each method in each class. To judge whether a class violates R1, the only way is compare R1's thresholds to collected information from P_2. For security reason, the protocols apply secure 2-party computation technique for comparison. Thus, no confidential information will be leaked and the two parties can still learn what kinds of defects exist in each class. P_1 and P_2 will call Threshold Evaluation routine to do the detection as following.

Threshold_Evaluation((P_1, {1500, 129, 100}, {$\wedge$, $\vee$}), (P_2, {1621, 145, 134}))

Then, P_1 and P_2 call Secure Comparison to compare each pair of integers.

Secure_Comparison((P_1, 1500), (P_2, 1621))

Secure_Comparison((P_1, 129), (P_2, 145))

Secure_Comparison((P_1, 100), (P_2, 134))

Next, P_1 combines these results together with operators as below.

$$1621 \geq 1500 \wedge 145 \geq 129 \vee 134 \geq 100$$

In this example, the output is true and then P_1 determines that class PrjInfos is a blob for secure comparison results judge that it violates the rule R1. This is just a round of an individual rule to detect a single class. Finally, each rule should be applied to detect every class and the total rounds will be up to $3 \times n$ (e.g. n classes in source models). After detection, P_1 and P_2 call Compares to compute fitness score of this rule set. Suppose P_1 expresses its detection results with an integer set as below.

$\{d_i\}$ ={{1, 0, 0}, {0, 1, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 1}}

$\{ed_i\}$ = {E(1), E(0), E(0), E(0), E(1), E(0), E(0), E(0), E(0), E(0), E(0), E(0), E(0), E(0), E(1)}

Each subset represents the defects of a class and the three integers in the subset denotes three types of defects. The integer is either 1 if the class contains this type of defect or 0 otherwise. Then, P_1 encrypts each integer and sends the sequence set $\{ed_i\}$ to P_2 who should also describe its example base with an integer set.

$\{e_i\}$ = {{1, 0, 0}, {0, 0, 0}, {1, 0, 0}, {0, 0, 0}, {0, 0, 1}}

Next, P_2 calculates the product of each pair of ed_i and e_i, then sums them up together to get the following result.

E(1) × E(1) = E(1 + 1) = E(2)

P_1 receives E(2) from P_2 and learns that the intersection is two after decryption. So the fitness of this rule set will be

f = (2/3 + 2/3)/2 = 0.67

Next, Genetic Programming process will perform crossover and mutation to generate new offsprings (new rules), and P_1 and P_2 will apply these new rules to detect source models again. For example, GP algorithm removed OR operation in R1, then a new rule R1' will be generated as below.

R1' : IF (LOCCLASS $\geq$ 1500 $\wedge$ LOCMETHOD $\geq$ 129) THEN defect = blob

Thus, for secure comparison algorithm, the input and output would be changed as the following. T={1500, 129}, S={LOCCLASS, LOCMETHOD}, O={$\wedge$}, C={LOCCLASS $\geq$1500, LOCMETHOD $\geq$ 129} and R=*C1* $\wedge$ *C2*.

If the new rule's fitness is better than previous ones, P_1 will record the new rule as current best solution. Finally, P_1 will get best detection rule set which is suitable to P_2's source models. In correction step, this rule set will be used to detect existing detects after each correction.

For correction, initially P_1 creates a n-dimensional refactoring solutions and sends them to P_2 for execution. In the above example, suppose the refactoring solutions are as below.

MoveMethod(getWebLink, PrjInfos, DeprecatedProjectExportData),
MoveAttribute(WebLink, PrjInfos, DeprecatedProjectExportData),
PushDownMethod(actionPerformed, GanttApplet, DeprecatedProjectExportData)

In the process, a fitness function is used to quantify the quality of the proposed refactorings, which checks to minimize the number of detected defects using the detection rules generated in detection step.

Next, P_1 will generate new refactoring solutions by Genetic Algorithm and request P_2 to execute them again. For instance, the mutation operator may change PushDownMethod to Movethod and a new set of refactorings will be as the following.

MoveMethod(getWebLink, PrjInfos, DeprecatedProjectExportData),
MoveAttribute(WebLink, PrjInfos, DeprecatedProjectExportData),
MoveRelation(GanttHTMLExport, getDescription, PrjInfos);

If the new solutions are better than all of others by fitness comparison, then P_1 saves them as current best solutions. At last, the process will select an optimal solution which fixes most defects. Then, P_2 can apply the solutions to correct defects in its source models.

6. EXPERIMENTAL RESULTS

This section will discuss how to test the secure protocols. For Generic Programming and Generic Algorithm in defect detection and correction, the work [13] has already verified their precision and recall rates, so the thesis do not plan to provide additional evaluations. This section will focus on the secure comparison protocol because it is the most time-consuming part in the whole process.

First, it is very important to compute the running time of secure comparison for it is important to know how it impacts the proposed approach. The cost of this algorithm highly depends on the bit size of encryption and decryption keys. Usually the keys are very large binary numbers, e.g. 512; 1024; 2048 bits, and people prefer to choose long bit keys for it is hard to be cracked. However, long keys really make the algorithm much inefficient and even unacceptable. The experiments show that for if two integers' comparison in the range of [0 - 512], the cost is 0.18s for 1024 bit keys and over 1.25s for 2048 bit keys. Because the protocols have to do comparison for every detection rule, the total running cost will be unbearable if long bit keys are used. In reality, it is safe enough to use 1024 bit keys to encrypt private data. Then in the following experiments, only 1024 bit keys are applied to the detection and correction algorithms.

Kessentini tests his approach with some open-source programs: GanttProject (Gantt for short) v1.10.2, Quick UML v2001, ArgoUML v0.19.8, and Xerces-J v2.7.0 as the Table 6.1 shown. The performance of Kessentini's approach is highly related to Generatic Algorithm which is actually unchanged in this scheme, whose running cost indeed depends on comparison times, the number of classes and rules. Thus, in the experimental settings, the thesis will pay more attention on measuring its performance under different size of source models rather than how many programs are used. Then the research work decides to use GanttProject and Xerces to do it for they are medium-sized programs and the results would clearly show the difference between two approaches. In addition, some classes or some defects are removed from the program to verify the performance of the secure approach in various scenarios.

Table 6.1. Program Statistics

Systems	Number of Classes	KLOC
GanttProject v1.10.2	245	31
Xerces-J v2.7.0	991	240
ArgoUML v0.19.8	1230	1160
Quick UML v2001	142	19

As previously mentioned, three types of defects will be analyzed. In a word, blobs are classes that do too much; spaghetti Code (SC) is code that does not use appropriate structuring mechanisms; finally, functional decomposition (FD) is code that is structured as a series of function calls. These represent different types of design risks. In the study, the thesis uses a cross validation procedure and one open source project is evaluated by using the remaining two systems as base of examples. For example, Xerces-J is analyzed using some defects examples from Gantt. The complete lists of metrics, used to generate rules, and applied refactorings can be found in [16].

Table 6.2 summarizes the testing results. In the experiments, SC represents the approach with secure comparison, and SC & SI means that both secure comparison and set intersection are applied to the approach. DET and COR are the abbreviations of detection and correction, respectively. The experimental results are not exciting because the proposed approach is much slower than the original one. The main reason is that the protocols have to do too many secure comparisons in order to preserve both parties' privacies. For example, the GanttProject program contains 245 classes and it is supposed that every individual rule has three operators in average, because a rule set includes three different rules to detect three types of defects, then the total number of comparison to evaluate a rule set is $245 \times 3 \times 3 = 2205$. It is mentioned that the average cost of running secure comparison once is 0.18s, thus the detection algorithm will cost $2205 \times 0.18 = 396.9$s which is very close to the experimental result. However, you may observe that there is no significant difference between the costs of SC and SC & SI, which is because the set intersection algorithm only runs once for an entire rule set. In the first scenario of GanttProjects, the execution rounds for SC and SC & SI are 2205 and 1, respectively, that

is why SI didn't cost too much time even if it still contains encryption and decryption algorithms.

The table also shows that even if some classes are removed from GanttProject to make it a small project, the running cost is still much higher than Kessentini's approach. To make it worse, the algorithm will cost similar time to do the detection even if all defects are deleted from the program, which is because the protocols cannot reduce the comparison times in detection step. For a larger program like Xerces-J, the detection process will cost nearly twenty minutes and it will spend almost fifty minutes to fix existing defects.

Table 6.2. Running Time Comparison(Seconds)

Systems	Classes	Blob	SC	FD	Original		SC		SC&SI	
					DET	COR	DET	COR	DET	COR
Gantt	245	10	14	9	7.76	16.87	350.46	704.21	359.07	801.57
Gantt	81	10	14	9	1.84	4.19	97.44	199.83	97.87	202.51
Gantt	81	0	0	0	1.64	0	96.26	0	96.74	0
Xerces	991	11	17	10	41.51	79.87	1219.21	2933.26	1321.04	3107.25

Another important issue is that the secure comparison and set intersection algorithms are implemented with C language because the approach integrates a C/C++ package, GMP into the developed algorithms for large integer computation, but the detection and correction algorithms are written with Java. Then, it will cost more time to call C routines in a Java program. Next step, all codes will be rewritten with C language and be integrated together, thus the algorithm will be much efficient than the current one. Moreover, in the future's work, it is possible to divide private data into different security levels and only encrypt data in high levels, then the algorithms' performance will be significantly improved and its expected running time might be reduced to the same magnitude as the original one.

7. CONCLUSION AND FUTURE WORK

This paper analyzes privacy issues in design defect detection and correction and then models TTP models in both defect detection and correction processes. In addition, it designs new secure protocols to allow a third party to perform such detection and corrections without leaking any private information. The main contribution is that the thesis propose a practical approach to replace TTPs and make it possible for a Consultant to offer detection and correction services while preserving both parties' privacy.

Moreover, the secure comparison is a time-consuming part in detection process, and the thesis analyzes its performance and compares running time of the approach with that of the original one. Experimental results prove the effectiveness of this approach. In the future, more defect detection and correction algorithms will be investigated and design common secure protocols may be designed, which are suitable to most popular detection and correction algorithms.

Finally, the proposed secure protocols may leak some private information compared to ideal models. In the following work, more effective and efficient SDDC protocols would be developed, which might be as secure as the ideal TTP models.

APPENDIX

A SIMPLE EXAMPLE OF SOURCE MODELS

Attribute(GanttCSVExport,prjInfos,PrjInfos,N,private);
Attribute(GanttProject,prjInfos,PrjInfos,N,public);
Attribute(GanttXFIGSaver,prjInfos,PrjInfos,N,private);
Attribute(PrjInfos,sDescription,String,N,public);
Attribute(PrjInfos,sOrganization,String,N,public);
Attribute(PrjInfos,sProjectName,String,N,public);
Attribute(PrjInfos,sWebLink,String,N,public);
Class(PrjInfos,N,N,public);
Method(NewProjectWizard,createNewProject,PrjInfos,
Y,N,N,public);
Method(PrjInfos,PrjInfos,N,N,N,N,public);
Method(PrjInfos,PrjInfos,N,Y,N,N,public);
Method(PrjInfos,getDescription,String,N,N,N,public);
Method(PrjInfos,getName,String,N,N,N,public);
Method(PrjInfos,getOrganization,String,N,N,N,public);
Method(PrjInfos,getWebLink,String,N,N,N,public);
Parameter(GanttCSVExport,GanttCSVExport,prjInfos,
PrjInfos,declaration);
Parameter(GanttHTMLExport,save,prjInfos,PrjInfos,declaration);
Parameter(GanttXFIGSaver,GanttXFIGSaver,prjInfos,
PrjInfos,declaration);
Parameter(PrjInfos,PrjInfos,sDescription,String,declaration);
Parameter(PrjInfos,PrjInfos,sOrganization,String,declaration);
Parameter(PrjInfos,PrjInfos,sProjectName,String,declaration);
Parameter(PrjInfos,PrjInfos,sWebLink,String,declaration);
Relation(GanttHTMLExport;save;getDescription,PrjInfos,N);
Relation(GanttHTMLExport;save;getName,PrjInfos,N);
Relation(GanttHTMLExport;save;getOrganization,PrjInfos,N);
Relation(GanttHTMLExport;save;getWebLink,PrjInfos,N);
Relation(GanttProject;getDescription;getDescription,PrjInfos,N);
Relation(GanttProject;getOrganization;getOrganization,PrjInfos,N);

Relation(GanttProject;getWebLink;getWebLink,PrjInfos,N);
Attribute(GanttApplet,button,JButton,N,private);
Attribute(GanttApplet,fileLocation,String,N,private);
Class(GanttApplet,N,N,public);
Generalisation(GanttApplet,JApplet);
Method(GanttApplet,GanttApplet,N,N,N,N,public);
Method(GanttApplet,actionPerformed,void,Y,N,N,public);
Method(GanttApplet,createContainer,Container,N,N,N,private);
Method(GanttApplet,init,void,N,N,N,public);
Method(GanttApplet,main,void,Y,N,static,public);
Parameter(GanttApplet,actionPerformed,e,ActionEvent,declaration);
Parameter(GanttApplet,actionPerformed,ganttFrameGanttProject,local);
Parameter(GanttApplet,actionPerformed,inSInputStream,local);
Parameter(GanttApplet,actionPerformed,urlURL,local);
Parameter(GanttApplet,createContainer,panelJPanel,local);
Parameter(GanttApplet,init,fileLocationParamString,local);
Parameter(GanttApplet,main,appletGanttApplet,local);
Parameter(GanttApplet,main,args,String[],declaration);
Parameter(GanttApplet,main,frameJFrame,local);
Relation(GanttApplet;actionPerformed;getCodeBase,Applet,N);
Relation(GanttApplet;actionPerformed;getInputStream,URLConnection,N);
Relation(GanttApplet;actionPerformed;openConnection,URL,N);
Relation(GanttApplet;actionPerformed;openXMLStream,
GanttProject,InputStream-String);
Relation(GanttApplet;actionPerformed;printStackTrace,Throwable,N);
Relation(GanttApplet;actionPerformed;setVisible,Window,boolean);
Relation(GanttApplet;actionPerformed;toString,URL,N);
Relation(GanttApplet;createContainer;add,Container,Component);
Relation(GanttApplet;createContainer;addActionListener,
AbstractButton,ActionListener);
Relation(GanttApplet;init;createContainer,GanttApplet,N);

Relation(GanttApplet;init;getParameter,Applet,String);
Relation(GanttApplet;init;setContentPane,JApplet,Container);
Relation(GanttApplet;main;createContainer,GanttApplet,N);
Relation(GanttApplet;main;pack,Window,N);
Relation(GanttApplet;main;setContentPane,JFrame,Container);
Relation(GanttApplet;main;setDefaultCloseOperation,JFrame,int);
Relation(GanttApplet;main;setVisible,Window,boolean);
Attribute(DeprecatedProjectExportData,myExportOptions,
GanttExportSettings,N,package);
Attribute(DeprecatedProjectExportData,myFilename,String,N,package);
Attribute(DeprecatedProjectExportData,myGanttChart,
GanttGraphicArea,N,package);
Attribute(DeprecatedProjectExportData,myProject,GanttProject,N,package);
Attribute(DeprecatedProjectExportData,myResourceChart,
ResourceLoadGraphicArea,N,package);
Attribute(DeprecatedProjectExportData,myTree,GanttTree,N,package);
Attribute(DeprecatedProjectExportData,myXslFoScript,String,N,package);
Class(DeprecatedProjectExportData,N,N,public);
Method(DeprecatedProjectExportData,DeprecatedProjectExportData,
N,Y,N,N,public);
Parameter(DeprecatedProjectExportData,DeprecatedProjectExportData,
myExportOptions,GanttExportSettings,declaration);
Parameter(DeprecatedProjectExportData,DeprecatedProjectExportData,
myFilename,String,declaration);
Parameter(DeprecatedProjectExportData,DeprecatedProjectExportData,
myGanttChart,GanttGraphicArea,declaration);
Parameter(DeprecatedProjectExportData,DeprecatedProjectExportData,
myProject,GanttProject,declaration);
Parameter(DeprecatedProjectExportData,DeprecatedProjectExportData,
myResourceChart,ResourceLoadGraphicArea,declaration);
Parameter(DeprecatedProjectExportData,DeprecatedProjectExportData,

myTree,GanttTree,declaration);
Parameter(DeprecatedProjectExportData,DeprecatedProjectExportData,
myXslFoScript,String,declaration);
Parameter(GanttProject,doExport,exportDataDeprecatedProjectExportData,
local);
Parameter(PDFExportProcessor,doExport,exportData,
DeprecatedProjectExportData,declaration);
Parameter(ProjectExportProcessor,doExport,exportData,
DeprecatedProjectExportData,declaration);
Relation(GanttProject;doExport;doExport,ProjectExportProcessor,
DeprecatedProjectExportData);

BIBLIOGRAPHY

[1] Rezgui A., Ouzzani M., Bouguettaya A. and Medjahed B, “Preserving privacy in web services,” In Proceedings of the the 4th international ACM workshop on Web information and data management, pp. 56-62, 2002

[2] R. Agrawal and R. Srikant, “Privacy-preserving data mining,” In Proceedings of the 2000 ACM SIGMOD Conference on Management of Data, ACM, pp. 14-19, 2000

[3] J. Camenisch and R. Chaabouni, “Efficient protocols for set membership and range proofs,” in Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology. Springer-Verlag Berlin, Heidelberg, pp. 234-252, 2008

[4] I. Damgard, M. Geisler and M. Kroigard, “Homomorphic encryption and secure comparison,” International Journal of Applied Cryptography, 1, no. 1, pp. 22-31, 2008

[5] K. Dhambri, H. A. Sahraoui and P. Poulin, “Visual detection of design Anomalies,” in CSMR. IEEE, pp. 279-283, 2008

[6] K. Erni and C. Lewerentz, “Applying design metrics to object-oriented frameworks,” in Proc. IEEE Symp. Software Metrics, IEEE Computer Society Press, 1996

[7] M. Fischlin, “A cost-effective pay-per-multiplication comparison method for millionaires,” Lecture Notes in Computer Science, pp. 457-471, 2001

[8] M. J. Freedman, K. Nissim and B. Pinkas, “Efficient private matching and set Intersection,” Lecture Notes in Computer Science, pp. 1-19, 2004

[9] M. Harman and J. A. Clark, “Metrics are fitness functions too,” in IEEE METRICS. IEEE Computer Society, pp. 58-69, 2004

[10] Reiko Heckel, “Algebraic graph transformations with application conditions,” M.S. thesis, TU Berlin, 1995

[11] Nigel Jefferies, Chris Mitchell and Michael Walker, “A proposed architecture for trusted third party services,” Cryptography: Policy and Algorithms 1029, pp. 98-104, 1996

[12] Y. Kataoka, M. D. Ernst, W. G. Griswold and D. Notkin, “Automated support for program refactoring using invariants,” in Proc. Intl Conf. Software Maintenance, pp. 736-743, 2001

[13] Marouane Kessentini, Wael Kessentini, Houari Sahraoui, Mounir Boukadoum and Ali Ouni, “Design defects detection and correction by example,” Program Comprehension (ICPC), 2011 IEEE 19th International Conference, pp. 81-90, 2011

[14] Marouane Kessentini, Houari Sahraoui and Mounir Boukadoum, “Model transformation as an optimization problem,” vol. 5301, pp. 159-173

[15] F. Khomh, S. Vaucher, Y.-G. Gueheneuc and H. Sahraoui, “A Bayesian approach for the detection of code and design smells,” In Proc. of the ICQS, 2009

[16] S. C. Kothari, L. Bishop, J. Sauceda and G. Daugherty, “A pattern based framework for software anomaly detection,” Software Quality Journal, 12, no. 2 June, pp. 99-120, 2004

[17] Hui Liu, Limei Yang, Zhendong Niu, Zhyi Ma and Weizhong Shao, “Facilitating software refactoring with appropriate resolution order of bad smells,” pp. 265-268

[18] R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws,” in Proc. of ICM04, 2004

[19] N. Moha, Y.-G. Gueheneuc, L. Duchien and A.-F. L. Meur, “A method for the specification and detection of code and design smells,” Transactions on Software Engineering (TSE), 2009

[20] A. E. Nergiz, M. E. Nergiz, T. Pedersen and C. Clifton, “Practical and secure integer comparison and interval check,” Social Computing (SocialCom), 2010 IEEE Second International Conference, pp. 791-799, 2010

[21] M. O’Keeffe and M. Cinneide, “Search-based refactoring: an empirical study,” Journal of Software Maintenance, pp. 345-364, 2008

[22] W. F. Opdyke, “Refactoring: A program restructuring aid in designing objectoriented application frameworks,” Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992

[23] Ali Ouni, Marouane Kessentini, Houari Sahraoui and Mounir Boukadoum, “Maintainability defects detection and correction: a multi-objective approach,” Automated Software Engineering vol. 20, pp. 47-79, 2012

[24] Vaidya, J. and Clifton, C., “Privacy preserving association rule mining in vertically partitioned data,” In Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD), pp. 639-644, 2002

VITA

Wenquan Wang was born in Heilongjiang, China which has been his home before moving to the United States for education. He earned his Bachelor's degree in Wireless Communication from Jilin University and Master's degree in Signal and Information Processing from Tianjin University, China. He is currently working towards completion of his Master's degree in Computer Science from the Missouri University of Science and Technology, Rolla in August 2013.