Università
Ca'Foscari
Venezia

Dottorato di ricerca in Informatica, XXIII ciclo
Scuola di dottorato in Scienze e Tecnologie
(A.A 2009-2010)

Tesi Di Dottorato
Settore Scientifico Disciplinare: INF/01

PROPERTY DRIVEN PROGRAM SLICING AND WATERMARKING IN THE ABSTRACT INTERPRETATION FRAMEWORK

Sukriti Bhattacharya, 955522

Tutore del dottorando

Prof. Agostino Cortesi

Direttore della Scuola di dottorato

Prof. Paolo Ugo

February 2011

Abstract

Abstract Interpretation theory formalizes the conservative approximation of the semantics of hardware and software computer systems. Abstract Interpretation approximates semantics according to some property of choice, this is formalized by choosing an appropriate abstract domain to use as abstraction of the semantics. A great variety of abstract domains can be formulated and the choice of domain offers a trade-off between precision and computational complexity. This thesis illustrates the instantiation of the Abstract Interpretation in the following scenarios,

- ☐ Program slicing.
- ☐ Watermarking relational databases.
- ☐ Watermarking program source code.

In this dissertation, a further refinement of the traditional slicing technique, called property driven program slicing is proposed by combining it with a static analysis in Abstract Interpretation framework. Very often, we are interested on a specific property of the variables in the slicing criterion, rather values. This is the case for instance, when dealing with non-interference in language-based security, where abstractions come into play in modeling the observational power of attackers. Therefore, when performing slicing, the abstract properties of variables and the abstract dependencies come into account. This approach of slicing, is not simply a pure generalization of a well-known technique, but provides new insights in links existing between different computer science fields.

Strengthening the ownership rights on outsourced relational database is very important in todays internet environment. In this context, the thesis introduces a distortion free watermarking technique that strengthen the verification of integrity of the relational databases by using a public zero-distortion authentication mechanism based on a Abstract Interpretation framework.

With the increasing amount of program source code which is distributed in the web, software ownership protection and detection is becoming an issue. In this scenario, a public key software watermarking (asymmetric watermarking) scheme is proposed which is similar in spirit to zero-knowledge proofs. The proposed approach of watermarking a source code is seen as an alternative to encryption as a way to support software authentication rather a tool for copyright protection.

Acknowledgements

The first person that I would like to thank is my supervisor Prof. Agostino Cortesi for his precious guide and encouragement over these years. I have learnt a lot from him. He taught me how to develop my ideas and how to be independent.

Thanks to my external reviewers Prof. Pierpaolo Degano and Prof. Nabendu Chaki for the time they spent on carefully reading the thesis and for their useful comments and suggestions.

My deep gratitude is devoted to all the members at the IT office specially, Giovanna Zamara, Fabrizio Romano and Gian-Luca Dei Rossi. A special thank goes to Laura Cappellesso for always having a solution to my problems.

I would like to thank all my colleagues and friends in Venice. Among them special thank goes to Arnab Chatterjee, Aditi Vidyarthi, Matteo Centenaro, Alberto Carraro, Matteo Zanioli, Luca Leonardi and Paolo Modesti, who wrote the *Sommario* of the thesis.

Special thank to the Faculty of Physics and Applied Computer Science, AGH University of Science and Technology, Cracow, Poland and Department of Computer Science & Engineering, University of Calcutta, India for allowing me a visiting researcher position during my Ph.D course. In particular, I would like to thank Prof. Khalid Saeed and Prof. Nabendu Chaki for their support and guidance.

Above all, I am grateful to my parents for their unconditional support and the patience and love over the years.

Sukriti Bhattacharya
Venice, February 2011

Contents

List of Figures

List of Tables

To my parents

Chapter 1

Introduction

1.1 Motivation

Static program analysis [99][42] is an automatic process that obtains properties of a program by analyzing its source code or object code. It has been widely used in optimizing compilers, for instance. But as software becomes more and more important in our society and as the reliability of software becomes safety-critical, static program analysis has become more important in different scenarios too.

Abstract Interpretation is a framework for program analysis introduced by Patrick and Radhia Cousot [43][43][40]. It is a formalized framework designed to deal with approximations, specially useful for the static analysis of programs. The basic idea is that a (possibly infinite) set of states is approximated by one abstract state. This abstraction is made based on some property of interest. The variables and functions of a program is also abstracted using the same property. This leads to a computable set of states which correctly approximates the program.

Abstract Interpretation simulates execution with an abstract semantics over a program. Different abstract semantics give information about different properties of the program. The different forms of abstraction over the semantics are called *abstract domains*.

Abstract Interpretation is a relatively new field of research, and it is being developed constantly. This framework is very general and can be applied to a great variety of problems. In fact, a lot of works have been achieved, either based on abstract interpretation or trying to improve different parts of the framework. Because

Abstract Interpretation is so general, there have been theoretical advances and researches to improve the possible applications of abstract interpretation. When we try to apply abstract interpretation techniques, we must consider the objects manipulated during the process. This question is crucial to the implementation because it determines the accuracy and the complexity of the computation. But in many cases, the objects that are available from classical computer science may be inadequate, because they don't take advantage of the possibility of approximation. Thus some specific objects and their manipulation have been defined to be used in Abstract Interpretation. Usually, they cannot represent any possible object but they come with approximation mechanisms. As more domains are being developed and more will be known about abstract domains in general, it will probably be even more useful in the future. The choice of abstract domain for a specific need will probably be easier as tools for custom made abstract domains probably will be richer. Abstract Interpretation has been used in different contexts as theory for abstraction. In this thesis we use Abstract Interpretation framework to describe (define) *program slicing* as *property driven program slicing* [13, 36], *relational database watermarking* as *relational data table abstraction* [14–16] and a novel *program source code watermarking* procedure based on *zero-knowledge proof system.*[17]

1.1.1 Program Slicing

Program slicing is the study of meaningful subprograms. Typically applied to the code of an existing program, a slicing algorithm is responsible for producing a program (or subprogram) that preserves a subset of the original programs behavior. A specification of that subset is known as a slicing criterion, and the resulting subprogram is a slice. Generally speaking, by applying a slicing technique on a program P with a slicing criterion C (i.e. a line of code in P), we get a program P' that behaves like P when focussing only on the variables in C. The sliced program P' is obtained through backward computation from P by removing all the statements that do not affect neither directly nor indirectly the values of the variables in C.

Slicing was introduced with the observation that "programmers use slices when debugging" [130]. Nevertheless, the application of program slicing does not stop there. Further applications include testing [13, 69], program comprehension [36],

model checking [93, 110], parallelisation [6, 131], software metrics [92, 109], as well as software restructuring and refactoring [53, 82, 89].

There can be many different slices for a given program and slicing criterion. Indeed, there is always at least one slice for a given slicing criterion: the program itself [129, 130]. However, slicing algorithms are usually expected to produce the smallest possible slices, as those are most useful in the majority of applications.

1.1.1.1 Property Driven Program Slicing

Very often, we are interested on a specific property of the variables in the slicing criterion, not on their exact actual values. For instance, in a recently proposed abstract interpretation based model of non-interference in language-based security, where abstractions come into play in modeling the observational power of attackers. Therefore, when performing slicing, the abstract properties of variables and the abstract dependencies are came into account. Again consider the case, when we are analyzing a program P and we want a variable x in P to have a particular property ρ. If we realize that, at a fixed program point, x does not have that desired property, we may want to understand which statements affect the computation of property ρ of x, in order to find out more easily where the computation was wrong. In this case we are not interested in the exact value of x, hence we may not need all the statements that a standard slicing algorithm would return. Therefore, the traditional value based static slicing is not adequate in this case. In this situation we would need a technique that returns the minimal amount of statements that actually affect the computation of a desired property of x. since, properties propagate less than values, some statements might affect the values but not the property. This can make debugging and program understanding tasks easier, since a smaller portion of the code has to be inspected when searching for some undesired behavior.

In this direction, our aim is to further refine the traditional slicing technique [129, 130, 132] by combining it with a static analysis in Abstract Interpretation [40, 42] based framework that looks for the statements affecting a fixed property of variables of interest rather values. This results into a deeper insight on the strong relation between slicing and property based dependency (abstract dependency).

Stmt. No.	P	$P_{(4,d)}$	$P^{sign}_{(4,d)}$
1	a=5;	a=5;	
2	b=2-a;	b=2-a;	
3	e=-b;		
4	c=a+b;	c=a+b;	
5	d=c+a*a+b*b-c+5;	d=c+a*a+b*b-c+5;	d=c+a*a+b*b-c+5;

TABLE 1.1: Value based vs property driven program slice

The idea can be summarized as follows. The resulting proposal is a fixed point computation where each iterate has two phases. First, the control flow analysis is combined with a static analysis in a Abstract Interpretation based framework. Hence, each program point of the program is enhanced with information about the abstract state of variables with respect to the property of interest. Then, a backward program slicing technique is applied to the augmented program exploiting the abstract dependencies. Though our approach of slicing, is not simply a pure generalization of a well-known technique, but provides new insights in links existing between different computer science fields.

Example 1.1. *Consider the program P in Figure 1.1. The value based traditional slicing algorithm with slicing criterion $(5, d)$ just removes the statement 3 from the initial program as it does not affect the slicing criterion at all. However, if we are interested in the sign of d at statement 5, the situation is quite different. The sign of d at 5 depends neither on a nor on b. Therefore, statement 1 and statement 2 are irrelevant. Again, unlike the standard approach there is no dependency between d and c, so in this case statement 4 is also irrelevant. Therefore, The property driven program slice $P^{sign}_{(5,d)}$ of program P with respect to slicing criterion $(5, d)$ and sign property contains less statements than it's value based slice $P_{(5,d)}$ with the same slicing criterion.*

1.1.2 Watermarking

According to Main and Oorschot [88], computer security can be classified as the following three types.

1. Data security, which is concerned with the confidentiality and integrity of data in transit and storage.

2. Network security, which aims to protect network resources, devices, and services.

3. Software security, which protects software from unauthorized access, modification, and tampering.

Due to the dramatically increased usage of the Internet, the ease on downloading seems to encourage people to use data tables and software source codes without authorization. In this literature we concentrate on data security and software security by means of watermarking. The idea is to provide a distortion-free watermarking algorithm for relational databases and software source code in Abstract Interpretation based framework.

Most of the research in the area of watermarking is aimed to a common goal. This goal is how to insert error or mark or data or formula or evidence and so on associated with a secret key known only by the data owner in order to prove the ownership of the data without losing its quality. In order to evaluate any watermark system, the following requirements are generally considered:

1. Readability: A watermark should convey as much information as possible, statistically detectable, enough to identify ownership and copyright unambiguously,

2. Security: Only authorized users can have access to the watermarking information,

3. Imperceptibility: The embedding process should not introduce any perceptible artifacts into original data and not degrade the perceive quality, and

4. Robustness: The watermark should be able to withstand various attacks while can be detected in the extraction process.

The extensive use of databases in applications is creating a need for protecting copyright of databases. There are a range of watermarking techniques available for protection of ownership, authentication and content integrity. Given the lead that multimedia watermarking research has over database watermarking research, one would hope that the standard multimedia techniques can be just carried over to the realm of relational databases. However, there exist some fundamental differences

between the characteristics of multimedia data and relational data, which make the adaptation of the known watermarking techniques not as easy as one would have desired.

Software watermarking is a method of software protection by embedding secret information into the text of software. We insert such secret information to claim ownership of the software. This enables the copyright holders to establish the ownership of the software by extracting this secret message from an unauthorized copy of this software when an unauthorized use of this software occurs.

Software watermarking can be regarded as a branch of digital watermarking, which started about 1954 [45]. Since the publication of a seminal work by Tanaka et al. in 1990 [123], digital watermarking has made considerable progress and become a popular technique for copyright protection of multimedia content which includes digital still images, audio and video sources. Digital watermarking aims at protecting a digital content from unauthorized redistribution and copying by enabling ownership provability over the content. These techniques have traditionally relied on the availability of a large "bandwidth" within which information can be indelibly and imperceptibly inserted while remaining certain essential properties of the original contents. Research on software watermarking started in the 1990s. The patent by Davidson and Myhrvold [47] presented the first published software watermarking algorithm.

1.1.2.1 Watermarking Relational Databases

Watermarking databases has unique requirements that differ from those required for watermarking digital audio or video products. Such requirements include: maintaining watermarked database usability, preserving database semantics, preserving database structure, watermarked database robustness, blind watermark extraction, and incremental updatability , among many other requirements. These fundamental differences between the characteristics of multimedia data and relational data, make the adaptation of the known watermarking techniques not as easy as one would have desired. Therefore, new watermarking techniques for databases have to be designed.

This is increasingly important in many applications where relational databases are publicly available on the Internet. For example, to provide convenient access

to information for users, governmental and public institutions are increasingly required to publish their data on the Internet [124]. The released data are usually in tabular form and in these cases, all released data are public; what is critical for the owner of the data is to make sure that the released data are not tampered, along with its ownership proof. To check the integrity of relational databases, an intuitive method is to use the traditional digital signature to detect the alterations of databases [8]. A hash value is computed over a relational database and then is signed with the owner's private key. The generated signature is then appended to the database or stored separately. Though this method is very simple, there are some problems with it,

- The signature can only be used to verify whether the database has been modified or not; it cannot be used to localize and recover the modifications.
- For a very large database, the failure of integrity verification will render the whole database useless.
- If there is a need to make some necessary modifications to the database, we have to compute a new signature and discard the previous one.
- It is computationally intensive to generate and verify the signatures.

Due to these problems, it is desirable that we have a fragile watermarking scheme for relational databases so that any modifications can be localized and detected. In addition, because the fragile watermarking scheme is private key based, its computational cost is obviously less than that of digital signature schemes.

We introduce a distortion free watermarking technique that strengthen the verification of integrity of the relational databases by using a public zero distortion authentication mechanism based on the Abstract Interpretation framework. The watermarking technique is partition based. The partitioning can be seen as a virtual grouping, which does not change neither the value of the tables elements nor their physical positions. We called this phase as *relation abstraction*, where the confidential *abstract relation* keeps the information of the partitions. Instead of inserting the watermark directly to the database partition, we treat it as an abstract representation of that concrete partition, such that any change in the concrete domain reflects in its abstract counterpart. This is the *partition abstraction* phase,

where the main idea is to generate a gray scale image associated with each partition as a watermark of that partition, that serves as tamper detection procedure, followed by employing a public zero distortion authentication mechanism to verify the ownership, the *authentication* phase.

1.1.2.2 Watermarking Program Source Code

With the increasing amount of program source code (most of the time in the form of bytecode) which is distributed in the web, software ownership protection and detection is becoming an issue. In particular, with multiple distributions of code, in order to prevent the risk of running fake programs, it is important to provide authentication proofs that do not overload the packages and that are easy to check. This is the aim of the so called Software Watermarking Techniques.

The actual creator of the software can established his authorship by software watermarking. Software watermarking refers to the process of embedding hidden information called watermark in a software source code by the creator of the software so that the authorship of the software can be proven where the presence of the secret data is demonstrated by a watermark recognizer. Since the watermark is secret, only the true author knows its value.

In general, it is not possible to devise watermarks that are immune to all conceivable attacks; it is generally agreed that a sufficiently determined attacker will eventually be able to defeat any watermark. In our vision, watermarking is a method that does not aim to stop piracy copying, but to check the ownership of the software. Therefore in our approach watermarking is seen as an alternative to encryption as a way to support software authentication rather a tool for copyright protection.

An important consideration in watermarking is protecting the watermark from the adversary. The adversary might tamper with the program and modify or completely remove the watermark so that the watermark recognition would fail.

We propose a public key software watermarking (asymmetric watermarking) scheme which is similar in spirit to zero-knowledge proofs introduced by Goldwasser, Micali, Rackoff [63]. Zero-knowledge proofs provide a solution in a situation where a prover wants to prove its knowledge of the truth of a statement to another party, called the verifier. However, the prover wants to convey its knowledge of the proof

without conveying the actual proof. The proof is provided by the interaction between the two parties, at the end of which the verifier is convinced of the provers knowledge of the proof. However, the verifier has not gained any knowledge in the interaction. In our case, we show the watermark by a zero knowledge proof.

Our main idea is to prove the presence of a watermark without revealing the exact nature of the mark. The embedding algorithm inserts a watermark W into the source code of the program P using the private key of the owner. In the verification process we established a protocol V that has access to the watermarked program P_W and to the corresponding public key. V proves the presence of watermark W in P_W without revealing the exact location of the watermark nor the nature of the watermark specified by private key.

1.2 Thesis Outline

This thesis is composed by 6 chapters. Each chapter provides a brief introduction explaining its contents, while the chapters describing original work have also a final discussion section about the problems addressed and the solutions proposed in the chapter. Chapter 2 provides notation and the basic algebraic notions that we are going to use in the following of the thesis, together with a brief introduction to abstract interpretation. Chapter 3 provides a refinement of the traditional slicing technique by combining it with a static analysis in abstract interpretation based framework, namely, *property driven program slicing*. In Chapter 4 a distortion free watermarking technique is introduced that strengthen the verification of integrity of the relational databases by using a public zero distortion authentication mechanism in a abstract interpretation based frame work. a public key software watermarking (asymmetric watermarking) scheme which is similar in sprit to zero-knowledge proofs is proposed in Chapter 5. Chapter 6 sums up the major contributions of this thesis and briefly describes future works that we would like to explore.

Chapter 2

Basic Notions

In this chapter, we introduce the basic algebraic notations and teminologies that we are going to use in the thesis. In Section 2.1 describes the mathematical background, recalling the basic notions of sets, functions and relations, followed by an overview of fixpoint theory [41] and a brief presentation of lattice theory [40, 61, 62], recalling the basic algebraic ordered structures and the definitions of upper closure operators and Galois connections and describing how these two notions are related to each other. Abstract Interpretation [40, 42] is introduced in Section 2.2, characterizing abstract domains in terms of both Galois connections and upper closure operators.The properties of soundness and completeness of abstract operators with respect to the corresponding concrete ones are also described. Dependence terminologies that are useful for discussing property driven program slicing is illustrated in Section 2.3. Section 2.4 describes the notion of zero-knowledge proofs by an example.

2.1 Mathematical Background

2.1.1 Sets

A set is a collection of objects. The notation $x \in C$ (x *belongs to* C) expresses the fact that x is an element of the set C. The *cardinality* of a set C represents the number of its elements and it is denoted as $|C|$. Let C and D be two sets. C is a *subset* of D, denoted $C \subseteq D$, if every element of C belongs to D. When $C \subseteq D$

and there exists at least one element of D that does not belong to C we say that C is properly contained in D, denoted $C \subset D$.Two sets C and D are equal, denoted $C = D$, if $C \subseteq D$ and $D \subseteq C$. Two sets C and D are different, denoted $C \neq D$, if there exists an element in C (in D) that does not belong to D (to C). The symbol $\emptyset$ denote the *empty set*, namely the set without any element. The *union* of C and D is defined as $C \cup D = \{x | x \in C \vee x \in D\}$.The *intersection* of C and D is defined as $C \cap D = \{x | x \in C \wedge x \in D\}$. Two sets C and D are *disjoint* if their *intersection* is the *empty set*, i.e., $C \cap D = \emptyset$. The set *difference*, i.e. the set of elements of C that do not belong to D is defined as $C \setminus D = \{x | x \in C \wedge x \notin D\}$. The powerset $\wp(C)$ of a set C is defined as the set of all possible subsets of C and is defined as $\wp(C) = \{D | D \subseteq C\}$.

2.1.2 Relations

Let x, y be two elements of a set C, we call *ordered pair* the element (x, y). This notion can be extended to the one of ordered $n-tuple$, with $n \geq 2 : (x_1, \ldots, x_n)$ is defined as, $(...((x_1, x_2), x_3)...)$. Given n sets $\{C_i\}_{1 \leq i \leq n}$, the cartesian product of the n sets C_i is the set of ordered n-tuples:

$$C_1 \times C_2 \times ...C_n = \{(x_1, x_2, ..., x_n) | \forall i : 1 \leq i \leq n : x_i \in C_i\}$$

We denote by C^n, the n^th cartesian self product of C.

Given two not empty sets C and D, any subset of the cartesian product $C \times D$ defines a *relation* between the elements of C and the elements of D. When $C = D$, any subset of $C \times C$ defines a binary relation on C. Given a relation $\mathcal{R}$ between C and D, i.e., $\mathcal{R} \subseteq C \times D$, and two elements $x \in C$ and $y \in D$, then $(x, y) \in \mathcal{R}$ and $x\mathcal{R}y$ are equivalent notations denoting that the pair (x, y) belongs to the relation $\mathcal{R}$, namely that x is in relation $\mathcal{R}$ with y. In the following we introduce two important classes of binary relations on a set C.

Definition 2.1. *(Equivalence relation)* An *equivalence relation* $\mathcal{R}$ on a set C is a subset of $C \times C$ which satisfies the three axioms below:

- reflexivity: $\forall x \in C : (x, x) \in \mathcal{R}$
- symmetry: $\forall x, y \in C : (x, y) \in \mathcal{R} \Rightarrow (y, x) \in \mathcal{R}$

- transitivity: $\forall x, y, z \in C : (x, y) \in \mathcal{R} \wedge (y, z) \in \mathcal{R} \Rightarrow (x, z) \in \mathcal{R}$

An *equivalence class* is a subset of C of the form $\{x \in C | x \mathcal{R} y\}$ where y is an element of C. Roughly, it contains all the elements of C which are equivalent to y.

Definition 2.2. *(Partial order)* A binary relation $\sqsubseteq$ on a set C is a *partial order* on C if the following properties hold:

- reflexivity: $\forall x \in C : x \sqsubseteq x$;
- antisymmetry: $\forall x, y \in C : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$;
- transitivity: $\forall x, y, z \in C : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$

A set with a partial order defined on it is called a *partially ordered set* or *poset* and is denoted as $\langle C, \sqsubseteq \rangle$.

2.1.3 Functions

Let C and D be two sets. A *function* f from C to D is a relation between C and D such that for each $x \in C$ there exists exactly one $y \in D$ such that $(x, y) \in f$, and in this case we write $f(x) = y$. Usually the notation $f : C \to D$ is used to denote a function from C to D, where C is the *domain* and D the *co-domain* of function f. The set $f = \{f(x) | x \in X\}$ is the image of $X \subseteq C$ under f. The image of the domain, i.e., $f(C)$, is called the *range* of f. If there exists an elements $x \in C$ such that the element $f(x)$ is not defined, we say that function f is *partial*, otherwise function f is said to be *total*. Given two sets C and D and function $f : C \to D$. Function f is said to be *injective* or *one-to-one* if for every $x, y \in C : f(x) = f(y) \Rightarrow x = y$. Thus, a function is injective if it maps distinct elements into distinct elements. Function f is *surjective* or *onto* if, $f(C) = D$. Thus, a function is *surjective* if every element of the co-domain is image of at-least one element of the domain. Function f is *bijective* if f is both *injective* and *surjective*. While Two sets are *isomorphic*, denoted $\cong$, if there exists a *bijection* between them. The *identity* function $f_{id} : C \to C$ that associates each element to itself, i.e., $\forall x \in C : f_{id}(x) = x$. The *composition* $g \circ f : C \to E$ of two functions $f : C \to D$ and $g : D \to E$, is defined as $g \circ f(x) = g(f(x))$.

2.1.4 Ordered Structures

Let us consider two elements x and y of a poset $\langle L, \sqsubseteq \rangle$. We say that x is covered by y in C, written $x \prec y$, if $x \sqsubseteq y$ and $\nexists z \in L : x \sqsubseteq z \sqsubseteq y$. Relation $\prec$ can be used to define a *Hasse diagram* for a finite ordered set C: the elements of C are represented by points in the plane, where x is drown above y if $x \sqsubseteq y$, and a line is drown from point x to point y precisely when $x \prec y$.

Following are definitions of some commonly used terms related to *partial order* $\langle L, \sqsubseteq \rangle$.

- $X \subseteq L$ has $l \in L$ as *lower bound* if $\forall l' \in X : l \sqsubseteq l'$.
- $X \subseteq L$ has $l \in L$ as *greatest lower bound* (*glb* or *inf* or *meet*) if l is a *lower bound* of X and $l_0 \sqsubseteq l$ whenever l_0 is another *lower bound* of X. It is represented by the operator $\sqcap$. We denote $glb(X)$ by $\sqcap X$.
- let l be the $glb(X)$, if l belongs to X, l is the minimal element of X. If $glb(L)$ exists it is called the *minimum* (or *bottom*) element of the poset L and it is usually denoted by the symbol $\bot$.
- $X \subseteq L$ has $l \in L$ as *upper bound* if $\forall l' \in X : l' \sqsubseteq l$.
- $X \subseteq L$ has $l \in L$ as *least upper bound* (*lub* or *sub* or *join*) if l is the *upper bound* of X and $l \sqsubseteq l_0$ whenever l_0 is another *upper bound* of X. It is represented by the operator $\sqcup$. We denote $lub(X)$ by $\sqcup X$.
- let l be the $lub(X)$, if l belongs to X, l is the maximal element of X. If $lub(X)$ exists it is called the *maximum* (or *top*) element of the poset L and it is usually denoted by the symbol $\top$.

Example 2.1. *Lets consider a partially ordered set* $\langle L, | \rangle$*; where* $L = \{1, 2, 3, 4, 6, 9, 36\}$ *under relation divides(*$|$*). It can be represented using Hasse diagram as shown in Figure 2.1. Here for* $X = \{6, 36\}$ *we get upper bound(X)* $= \{3, 6\}$ *and lower bound(X)* $= \{1\}$*. Hence glb(X) = 1 and lub(X)= 6. Hence it is clear from this example that glb and lub for a given subset* X *need not be present in* X*.*

Definition 2.3. *(Chain)* Let $\langle L, \sqsubseteq \rangle$ be a poset. $X \subseteq L$ is a *chain* if $\forall x, y \in X : x \sqsubseteq y \vee y \sqsubseteq x$. Hence, a chain is a totally ordered set. Figure 2.2 shows a poset and some chains in it.

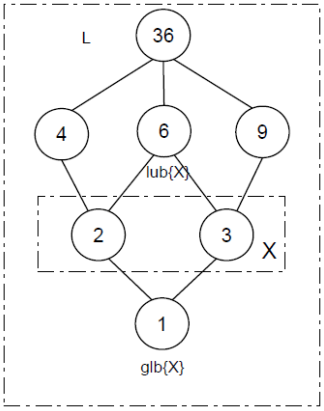


FIGURE 2.1: Complete lattice

Definition 2.4. *(Direct set)* A poset L is a *direct set* if each non-empty finite subset of L has least upper bound in L. A typical example of a direct set is a chain.

Definition 2.5. *(Complete partial order)* A *complete partial order* (or *cpo*) is a poset $\langle L, \sqsubseteq \rangle$, such that there is a *bottom* element $\bot$ and for each *direct set* D in L, there exists $\sqcup D$.

It is clear that every finite poset is a *cpo*. Moreover, it holds that a poset L is a *cpo* if and only if each *chain* in L has *least upper bound*.

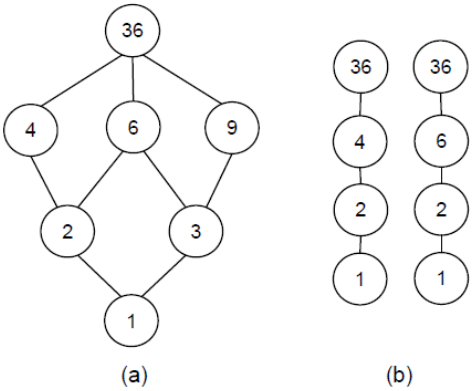


FIGURE 2.2: (a)Poset(b)Example of chains in poset(a)

Definition 2.6. *(Complete lattice)* A *complete lattice* $(L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ is a partial ordered set (L, $\sqsubseteq$) such that every subsets of L, has a *least upper bound* as well as a *greatest lower bound*.

- The greatest element $\top = \sqcap\emptyset = \sqcup L$

- The least element $\bot = \sqcap L = \sqcup \emptyset$

Definition 2.7. *(Moore Family)* Let L be a complete lattice. The subset $X \in C$ is a *Moore family* of L if $X = \mathcal{M}(X) = \{\sqcap S | S \subseteq X\}$, where $\sqcap \emptyset = \top \in \mathcal{M}(X)$.

In fact $\mathcal{M}(X)$ is the smallest (with respect to set inclusion) subset of L containing X.

Definition 2.8. *(Ascending chain condition)* A poset $\langle C, \sqsubseteq \rangle$ satisfies the *ascending chain condition* (ACC) if for each increasing sequence $x_1 \sqsubseteq x_2 \sqsubseteq ... \sqsubseteq x_n \sqsubseteq ...$ of elements of C, there exists an index k such that $\forall h \geq k$ it holds $x_h = x_k$ i.e $x_k = x_{k+1} = ...$

It is clear that the ordered set of even numbers $\{n \in \mathbb{N} \mid n \ \ mod \ \ 2 = 0\}$ does not satisfy the ascending chain condition, since the ascending chain of even numbers does not converge. A poset satisfying the *descending chain condition* (DCC) is dually defined as a poset without infinite descending chains.

2.1.5 Functions on Domains

Let $\langle C, \sqsubseteq_1 \rangle$ and $\langle D, \sqsubseteq_2 \rangle$ be two posets, and consider a function $f : C \to D$, now let us consider the following definitions:

Definition 2.9. *(monotone)* f is *monotone* (or *order preserving*) if for each $x, y \in C$ such that $x \sqsubseteq_1 y$ we have that $f(x) \sqsubseteq_2 f(y)$.

Definition 2.10. *(order embedding)* f is *order embedding* if $\forall x, y \in C$ we have that $x \sqsubseteq_1 y \Leftrightarrow f(x) \sqsubseteq_2 f(y)$.

Definition 2.11. *(order isomorphism)* f is an *order isomorphism* if f is order embedding and surjective.

The continuous and additive functions are particularly important when studying program semantics.

Definition 2.12. *(continuous)* Given two *cpo* C and E, a function $f : C \to E$ is *(Scott)- continuous* if it is *monotone* and if it preserves the limits of direct sets, i.e. if for each direct set D of C, we have $f(\sqcup_C D) = \sqcup_E f(D)$.

Co-continuous functions can be defined dually.

Definition 2.13. *(additive)* Given two *cpo* C and D, a function $f : C \to D$ is (*completely*) *additive* if for each $X \subseteq C$, we have that $f(\sqcup_C X) = \sqcup_D f(X)$.

Hence, an additive function preserves the limits (*lub*) of all subsets of C ($\emptyset$ is included), meaning that an *additive* function is also *continuous*. The notion of *co-additive* functions is dually defined.

2.1.6 Fixed Points

Definition 2.14. *(Fixpoint)* Consider a monotone function $f : L_1 \to L_2$ on a complete lattice $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. A *fixed point* of f is an element $l \in L$ such that $f(l) = l$.

$$Fix(f) = \{l | f(l) = l\}$$

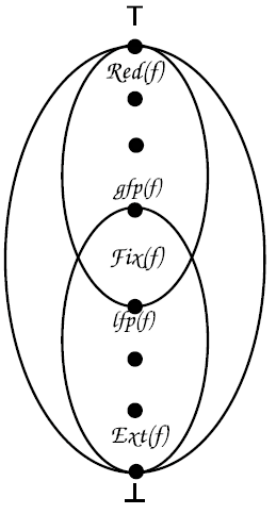


FIGURE 2.3: Structure of fixed points of f

The semantics of programming languages can be always formulated in a fixpoint form. If f is defined over a partial order $\langle L, \sqsubseteq \rangle$ (Figutre 2.3), then an element $l \in L$ is

- Fix(f) $= \{l \in L \mid f(l) = l\}$ set of fixed points.
- a *pre-fixpoint* if $l \sqsubseteq f(l)$

- a *post-fixpoint* if $f(l) \sqsubseteq l$
- the *least fixpont* ($\text{lfp}(f)$)if $\forall l' \in L : l' = f(l') \Rightarrow l \sqsubseteq l'$
- the *greatest fixpoint*($\text{gfp}(f)$) if $\forall d' \in L : l' = f(l') \Rightarrow l' \sqsubseteq d$
- $\text{Ref}(f) = \{l | f(l) \sqsubseteq l\}$ is the set of elements upon which f is *reductive*.
- $\text{Ext}(f) = \{l | f(l) \sqsupseteq l\}$ is the set of elements upon which f is *extensive*.
- $\text{Fix}(f) = \text{Red}(f) \cap \text{Ext}(f)$.

By Tarski's theorem, we have

$$\text{lfp}(f) \stackrel{\text{def}}{=} \sqcap \text{Fix}(f) = \sqcap \text{Ref}(f) \in \text{Fix}(f)$$
$$\text{Fix}(f) \stackrel{\text{def}}{=} \sqcup \text{Fix}(f) = \sqcup \text{Ext}(f) \in \text{Fix}(f)$$

2.1.7 Closure Operators

Closure operator is very important when dealing with abstract interpretation.

Definition 2.15. *Closure* An upper *closure operator*, or simply a *closure*, on a *poset* $\langle C, \sqsubseteq \rangle$, is an operator $\rho : C \to C$ that is:

- extensive: $\forall x \in C : x \sqsubseteq \rho(x)$
- monotone: $\forall x, y \in C : x \sqsubseteq y \Rightarrow \rho(x) \sqsubseteq \rho(y)$
- idempotent: $\forall x \in C : \rho(\rho(x)) = \rho(x)$

Example 2.2. *Function $f : C \to C$ in Figure 2.4(a) is an upper closure operator while function $g : C \to C$ in 2.4(b) is not since it is not idempotent.*

Let *uco(C)* denote the set of all *upper closures* operators of domain C. If (L, $\sqsubseteq$, $\sqcup$, $\sqcap$, $\top$, $\bot$) is a complete lattice, then for each *closure operator* $\rho \in uco(C)$

$$\rho(c) = \sqcap \{x \in C | x = \rho(x), c \sqsubseteq x\}$$

meaning that the image of an elements c through ρ is the minimum *fixpoint* of ρ greater than c. Moreover, ρ is uniquely determined by its image $\rho(C)$, that is the set of its *fixpoints* $\rho(C) = Fix(\rho)$.

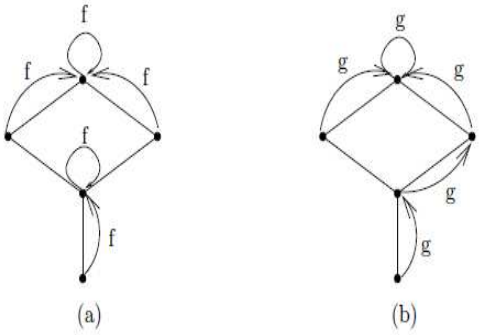


FIGURE 2.4: f is an upper closure operator while g is not

2.1.8 Galois Connections

Definition 2.16. *(Galois connection)* Two posets $\langle C, \sqsubseteq_1 \rangle$ and $\langle L, \sqsubseteq_2 \rangle$ and two monotone functions $\alpha : C \to L$ and $\gamma : L \to C$ such that:

- $\forall c \in C : c \sqsubseteq_1 \gamma(\alpha(c))$
- $\forall l \in L : \alpha(\gamma(l)) \sqsubseteq_2 l$

form a Galois connection, equivalently denoted by (C, α, γ, L).

A *Galois connection* (C, α, γ, L) where $\forall l \in L : \alpha(\gamma(l)) = l$ is called a *Galois insertion*.

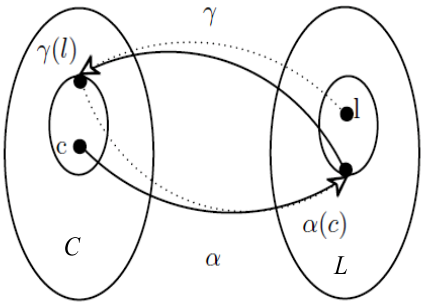


FIGURE 2.5: Galois connection

Given a *Galois connection* (C, α, γ, L) where $\langle C, \sqsubseteq_1 \rangle$ and $\langle L, \sqsubseteq_2 \rangle$ are posets, we have that:

- if C has a *bottom* element $\bot_1$, then L has *bottom* element $\alpha(\bot_1)$.
- dually, if L has *top* element $\top_2$, then C has *top* element $\gamma(\top_2)$.

- $\alpha \circ \gamma \circ \alpha = \alpha$ and $\gamma \circ \alpha \circ \gamma = \gamma$.
- if (L, α', γ', E) is a *Galois connection*, then $(C, \alpha' \circ \alpha, \gamma' \circ \gamma, E)$ is a *Galois connection*, namely it is possible to compose *Galois connections*.
- if (C, α, γ, L) is a *Galois insertion* and C is a *complete lattice*, then L is a *complete lattice*.
- α is *surjective* if and only if γ is *injective* if and only if (C, α, γ, L) is a *Galois insertion*, i.e. a *Galois insertion* between two complete lattices C and L is fully specified by a *surjective* and *additive* map $\alpha : C \to L$ or by an *injective* and *co-additive* map $\gamma : L \to C$.

2.1.8.1 Galois Connections and Closure Operators

Closure operators $\rho \in uco(C)$, being equivalent to *Galois connections*, have properties (*monotonicity*, *extensivity* and *idempotency*) that well fit the abstraction process. The monotonicity ensures that the approximation process preserves the relation of being more precise than. If a concrete element c_1 contains more information than a concrete element c_2, then after approximation we have that $\rho(c_1)$ is more precise than $\rho(c_2)$ (*monotonicity*). Approximating an object means that we could loose some of its properties, therefore it is not possible to gain any information during approximation. Hence, when approximating an element we obtain an object that contains at most the same amount of information of the original object. This is well expressed by the fact that the closure operator is *extensive*. Finally, we have that the approximation process looses information only on its first application, namely if the approximated version of the object c is the element a, then approximating a we obtain a. Meaning that the approximation function as to be *idempotent*. Hence, it is possible to describe abstract domains on C in terms of both *Galois connections* and upper closure operators [42]. Given a *Galois connection* (C, α, γ, L) it can be proved that the map $\gamma \circ \alpha$ is an *upper closure* on C, i.e., $\gamma \circ \alpha \in uco(C)$. If C is a *complete lattice* then $\gamma(L)$ is a *Moore family* of C and given a *poset* C and a *closure* $\rho \in uco(C)$ then $(C, \rho, \lambda x.x, \gamma(\alpha(C)))$ defines a *Galois insertion*. Thus, the notions of *Galois insertion* and *closure operators* are equivalent. This holds also for Galois connections up to reduction.

2.2 Abstract Interpretation

According to a widely recognized definition [43]: *Abstract interpretation is a general theory for approximating the semantics of discrete dynamic systems. Abstract interpretation* is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices. It can be viewed as a partial execution of a computer program which gains information about its semantics (e.g. control structure, flow of information) without performing all the calculations.

Abstract interpretation formalizes the approximation corresponding between the concrete semantics of a syntactically correct program and abstract semantics which is a safe approximation on the concrete semantics. Let $\mathcal{S}$ denotes a formal definition of the semantics of programs in $\mathbb{P}$ written in a certain programming language, and let $\mathcal{C}$ be the semantic domain on which $\mathcal{S}$ is computed. Let us denote with $\mathcal{S}^{\#}$ an abstract semantics expressing an approximation of the concrete semantics $\mathcal{S}$. The definition of the abstract semantics $\mathcal{S}^{\#}$ is given by the definition of the concrete semantics $\mathcal{S}$ where the domain $\mathcal{C}$ has been replaced by an approximated semantic domain $\mathcal{A}$ in Galois connection with $\mathcal{C}$, i.e., $(\mathcal{C}, \alpha, \gamma, \mathcal{A})$. Then, the abstract semantics is obtained by replacing any function $\mathcal{F} : \mathcal{C} \to \mathcal{C}$, used to compute $\mathcal{S}$, with an approximated function $\mathcal{F}^{\#} : \mathcal{A} \to \mathcal{A}$ that correctly mimics the behaviour of $\mathcal{F}$ in the domain properties expressed by $\mathcal{A}$.

2.2.1 Concrete vs Abstract Domains

The concrete program semantics $\mathcal{S}$ of a program $\mathbb{P}$ is computed on the so-called *concrete domain*, i.e., the *poset* of mathematical objects $\langle \mathcal{C}, \sqsubseteq_{\mathcal{C}} \rangle$ on which the program runs. The ordering relation encodes relative precision: $c_1 \sqsubseteq_{\mathcal{C}} c_2$ means that c_1 is a more precise (*concrete*) description than c_2. For instance, the *concrete domain* for a program with integer variables is simply given by the powerset of integer numbers ordered by subset inclusion $\langle \wp(\mathbb{Z}), \subseteq \rangle$.

Approximation is encoded by an *abstract domain* $\langle \mathcal{A}, \sqsubseteq_{\mathcal{A}} \rangle$, which is a poset of *abstract* values that represent some approximated properties of *concrete* objects. Also in the *abstract domain*, the ordering relation models relative precision: $a_1 \sqsubseteq_{\mathcal{A}} a_2$ means that a_1 is a better approximation (i.e., more precise) than a_2.

Example 2.3. (The abstract domain of intervals) *The set of intervals over integers is naturally equipped with a lattice structure, induced by the usual order on integers, extended to infinite positive and negative values in order to get completeness. When considering the concrete domain of the powerset of integers a non trivial and well known abstraction is given by the abstract domain of intervals, here denoted by* $\langle Int, \sqsubseteq_{Int} \rangle$ *[99]. The lattice* $(Int, \sqsubseteq_{Int}, \sqcup_{Int}, \sqcap_{Int})$ *of intervals is defined as,*

- *a base set* $Int = \{[a,b] | a,b \in \overline{\mathbb{Z}}, a \leq b\} \cup \bot$ *where* $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, +\infty\}$,
- *a partial order* $\sqsubseteq_{Int}$ *which is the least relation satisfying the following rules* $\frac{I \in Int}{\bot \sqsubseteq_{int} I}$ $\quad$ $\frac{c \leq a \quad b \leq d \quad a,b,c,d \in \overline{\mathbb{Z}}}{[a,b] \sqsubseteq_{Int} [c,d]}$
- *meet operator is defined by,*
  - $I \sqcup_{Int} \bot = I$, $\forall I \in Int$
  - $\bot \sqcup_{Int} I = I$, $\forall I \in Int$
  - $[a,b] \sqcup_{Int} [c,d] = [min(a,c), max(b,d)]$
- *join operators is defined by,*
  - $I \sqcap_{Int} \bot = I$, $\forall I \in Int$
  - $\bot \sqcap_{Int} I = I$, $\forall I \in Int$
  - $[a,b] \sqcap_{Int} [c,d] = [max(a,c), min(b,d)]$
- *The bottom and top elements are* $\bot_{Int} = \bot$ *and* $\top_{Int} = \{-\infty, +\infty\}$, *respectively.*

Figure 2.6 represents the abstract domain of intervals. $(\wp(\mathbb{Z}), \alpha_{Int}, \gamma_{Int}, Int)$ *is a Galois insertion where the abstraction* $\alpha_{Int} : \wp(\mathbb{Z}) \to Int$ *and concretization* $\gamma_{Int} : Int \to \wp(\mathbb{Z})$ *maps are defined as follows, let* $a, b \in \mathbb{Z}$ *then:*

$$\alpha_{Int}(S) = \begin{cases} \bot & if \quad S = \emptyset \\ [a,\ b] & if \quad min(S) = a \wedge max(S) = b \\ (-\infty, b] & if \quad \nexists min(S) \wedge max(S) = b \\ [a, +\infty) & if \quad min(S) = a \wedge \nexists max(S) \\ (-\infty, +\infty) & if \quad \nexists min(S)\ \nexists max(S) \end{cases}$$

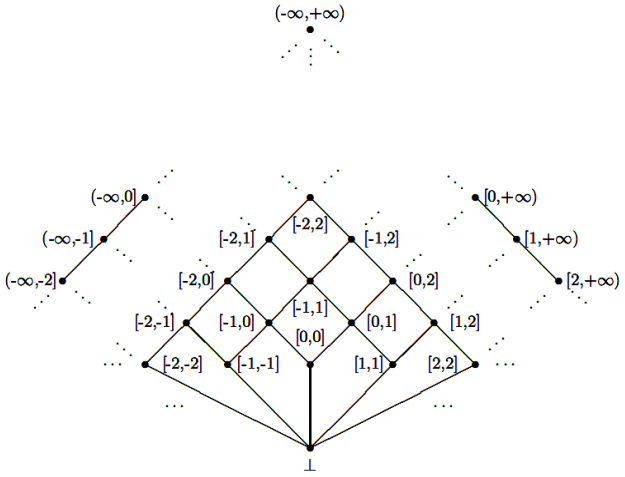


FIGURE 2.6: The *Interval* abstract domain

$$\gamma_{Int}(I) = \begin{cases} \emptyset & if \quad I = \bot \\ \{z \in \mathbb{Z} | a \leq z \leq b\} & if \quad I = [a, b] \\ \{z \in \mathbb{Z} | z \leq b\} & if \quad I = (-\infty, b] \\ \{z \in \mathbb{Z} | z \geq a\} & if \quad I = [a, +\infty) \\ \mathbb{Z} & if \quad I = (-\infty, +\infty) \end{cases}$$

For example, the set $\{2, 5, 8\}$ *is abstracted in the interval* $[2, 8]$*, while the infinite set* $\{z \in \mathbb{Z} | z \geq 10\}$ *is abstracted in the interval* $[10, +\infty)$*.*

Example 2.4. (The abstract domain of sign) *Figure 2.7 represents the abstract domain of sign.* $(\wp(\mathbb{Z}), \alpha_{sign}, \gamma_{sign}, \wp(\mathfrak{sign}))$ *is a Galois insertion where* $\forall z \in \mathbb{Z}$*,* $\mathfrak{sign} : \mathbb{Z} \to sign$ *is specified by,*

$$\mathfrak{sign}(z) = \begin{cases} - & if \quad z < 0 \\ 0 & if \quad z = 0 \\ + & if \quad z > 0 \end{cases}$$

$\forall C \in \wp(\mathbb{Z})$ *and* $\forall A \in \wp(\mathfrak{sign})$*, the abstraction* $\alpha_{sign} : C \to A$ *and concretization* $\gamma_{sign} : A \to C$ *maps are defined as follows,*

$$\alpha_{sign}(C) = \{\mathfrak{sign}(z) | z \in C\}$$
$$\gamma_{sign}(A) = \{z \in C | \mathfrak{sign}(z) \in A\}$$

Example 2.5. (The abstract domain of parity) *Fig. 8 represents the abstract domain of parity.* $(\wp(\mathbb{Z}), \alpha_{parity}, \gamma_{parity}, \wp(\mathfrak{parity}))$ *is a Galois insertion where* $\forall z \in \mathbb{Z}$*,* $\mathfrak{parity} : \mathbb{Z} \to parity$ *is specified by,*

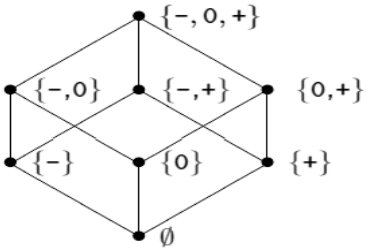


FIGURE 2.7: The *Sign* abstract domain

$$\mathfrak{parity}(z) = \begin{cases} Even & if \quad z\%2 = 0 \\ Odd & Otherwise \end{cases}$$

$\forall C \in \wp(\mathbb{Z})$ *and* $\forall A \in \wp(\mathfrak{parity})$*, the abstraction* $\alpha_{parity} : C \to A$ *and concretization* $\gamma_{parity} : A \to C$ *maps are defined as follows,*

$$\alpha_{parity}(C) = \{\mathfrak{parity}(z) | z \in C\}$$
$$\gamma_{sign}(A) = \{z \in C | \mathfrak{parity}(z) \in A\}$$

2.2.2 Abstract operations

2.2.2.1 Soundness

A concrete semantic operation must be approximated on some abstract domain $\mathcal{A}$ by a sound abstract operation $f^{\#} : \mathcal{A} \to \mathcal{A}$. This means that $f^{\#}$ must be a correct approximation of f in A: for any $c \in \mathcal{C}$ and $a \in \mathcal{A}$, if a approximates c then $f^{\#}(a)$ must approximates $f(c)$. This is therefore encoded by the condition:

$$\forall c \in C : \alpha(f(c)) \sqsubseteq_{\mathcal{A}} f^{\#}(\alpha(c)) \tag{2.1}$$

Soundness can be also equivalently stated in terms of the concretization map:

$$\forall a \in A : f(\gamma(a)) \sqsubseteq_{\mathcal{C}} \gamma(f^{\#}(a)) \tag{2.2}$$

In Figure 2.8 we have a graphical representation of soundness. In particular, Figure 2.8(a) refers to the condition $\alpha \circ f(x) \sqsubseteq_{\mathcal{A}} f^{\#} \circ \alpha(x)$, which compares the

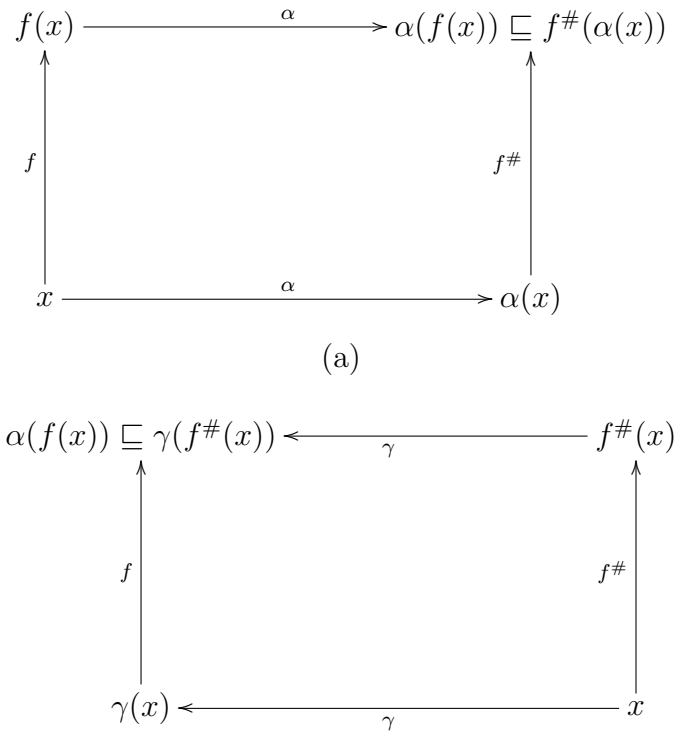


FIGURE 2.8: Soundness

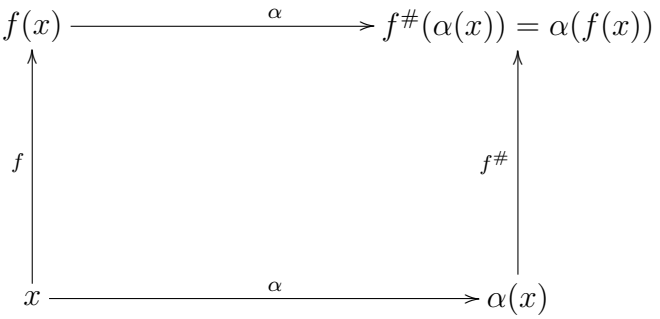
computational process in the abstract domain, while 2.8(b) refers to the condition $f \circ \gamma(x) \sqsubseteq_{\mathcal{C}} \gamma o f^{\#}(x)$, which compares the results of the computations on the concrete domain. Given a concrete operation $f : \mathcal{C} \to \mathcal{C}$, we can order the correct approximations of f with respect to $(\mathcal{C}, \alpha, \gamma, \mathcal{A})$: let $f_1^{\#}$and $f_2^{\#}$ be two correct approximations of f in $\mathcal{A}$, then $f_1^{\#}$ is a better approximation of $f_2^{\#}$ if $f_1^{\#} \sqsubseteq f_2^{\#}$. It is well known that, given a concrete function $f : \mathcal{C} \to \mathcal{C}$ and a Galois connection $(\mathcal{C}, \alpha, \gamma, \mathcal{A})$, there exists a best correct approximation of f on $\mathcal{A}$, usually denoted as $f^{\mathcal{A}}$. In fact, it is possible to show that $\alpha \circ f \circ \gamma : \mathcal{A} \to \mathcal{A}$ is a correct approximation of f on $\mathcal{A}$, and that for every correct approximation $f^{\#}$of f we have that: $\forall x \in A : \alpha(f(\gamma(x))) \sqsubseteq_{\mathcal{A}} f^{\#}(x)$. Observe that the definition of best correct approximation only depends upon the structure of the underlying abstract domain, namely the best correct approximation of any concrete function is uniquely determined by the Galois connection $(\mathcal{C}, \alpha, \gamma, \mathcal{A})$.

Example 2.6. (Soundness) *A (unary) integer squaring operation (sq) on the concrete domain* $\wp(\mathbb{Z})$ *is given by* $sq(X) = \{x^2 \in \mathbb{Z} | x \in X\}$. *A correct approximation* $sq^{\#}$ *of* sq *on the abstract domain Sign can be defined as follows:* $sq^{\#}(\top) = \top$, $sq^{\#}(\bot) = \bot$, $sq^{\#}(0+) = 0+$, $sq^{\#}(0-) = 0+$ *and* $sq^{\#}(0) = 0$.

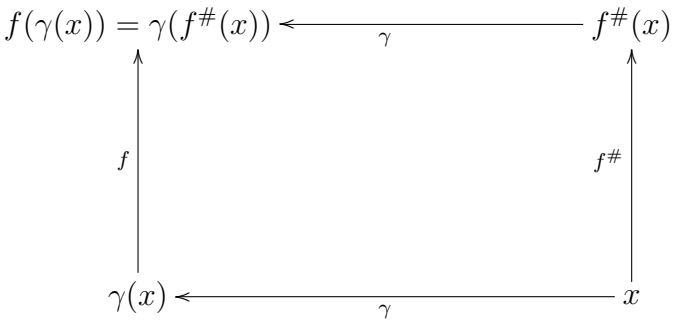
It is clear that this provides a sound approximation of the square function, that is $\forall x \in Sign : sq(\gamma(x)) \subseteq sq^{\#}(\gamma(x))$.

2.2.2.2 Completeness

When soundness is satisfied with equality, we say that the abstract function is a *complete* approximation of the concrete one. The equivalent *soundness* conditions (1) and (2) introduced above can be strengthened to two different (i.e., incomparable) notions of *completeness*.



(a) Backward completeness



(a) Forward completeness

FIGURE 2.9: Completeness

Given a Galois connection $(\mathcal{C}, \alpha, \gamma, \mathcal{A})$ and a concrete function $f : \mathcal{C} \to \mathcal{C}$, and an abstract function $f^{\#} : \mathcal{A} \to \mathcal{A}$, then:

- if $\alpha \circ f = f^{\#} \circ \alpha$, the abstract function $f^{\#}$ is *backward-complete* for f.
- if $f \circ \gamma = \gamma \circ f^{\#}$, the abstract function $f^{\#}$ is *forward-complete* for f.

Backward completeness considers abstractions on the output of operations while *forward completeness* considers abstractions on the input to operations. Figure 2.9(a) provides a graphical representation of *backward completeness*, while Figure 2.9(b) represents the *forward completeness* case. While any abstract domain $\mathcal{A}$ induces the so-called canonical best correct approximation, not all abstract domains induce a *backward (forward)-complete* abstraction. However, if there exists a complete function for f on the abstract domain $\alpha((C))$, then $\alpha \circ f \circ \gamma$ is also complete and viceversa. This means that it is possible to define a complete function for f on $\alpha(C)$ if and only if $\alpha \circ f \circ \gamma$ is complete [61].

Example 2.7. (Completeness) *A (unary) integer squaring operation (sq) on the concrete domain* $\wp(\mathbb{Z})$ *is given by* $sq(X) = \{x^2 \in Z | x \in X\}$. *A correct approximation* $sq^\#$ *of sq on the abstract domain Sign can be defined as follows:* $sq^\#(\top) = \top$, $sq^\#(\bot) = \bot$, $sq^\#(0+) = 0+$, $sq^\#(0-) = 0+$ *and* $sq^\#(0) = 0$. $sq^\#$ *is backward complete for sq on Sign while it is not forward complete because* $sq(\gamma(0+)) = \{x^2 \in \mathbb{Z} | x > 0\} \subsetneq \{x \in \mathbb{Z} | x > 0\} = \gamma(sq^\#(0+))$.

2.3 Dependence Terminologies

Dependence analysis is the process of determining a program's dependences, combining traditional control flow analysis and dataflow analysis. Program dependences are relationships, holding between program statements, that can be determined from a program text and used to predict aspects of the program execution behavior. There are two basic types of program dependences: *control dependences*, which are features of a programs control structure, and *data flow dependences*, which are features of a programs use of variables.

The following definitions are language independent. They are used to establish a common terminology to be used to define dataflow based property driven program slicing.

Definition 2.17. *(Directed graph)* A Directed graph or digraph G is a pair *(N, E)*, where N is a finite, nonempty set called nodes, and E is a subset of $N \times N - \{(n, n) | n \in N\}$, called edges of G, respectively. Given an edge $(n_i, n_j) \in E$, n_i is said to be predecessor of n_j, and n_j is said to be successor of n_i. *PRED(n)* and *SUCC(n)* are the set of the predecessors and the set of the successors of a node n, respectively. The in-degree of a node n, denoted $in(n)$, is the number

of predecessors of n, while the out-degree of n, denoted $out(n)$, is the number of successors of n. A walk W in a digraph G is a sequence of nodes $n_1, n_2...n_k$ such that $k \geq 0$ and $(n_i, n_{i+1}) \in E$ for $i = 1, 2, k - 1$, where k is the length of W. If W is nonempty (the length is not zero) then it is called a $n_1 - n_k$ walk.

Definition 2.18. *(A control-flow graph)* A control-flow graph G is a directed graph that satisfies each of the following conditions:

1. The maximum out-degree of the nodes of G is at most two.
2. G contains two distinguished nodes: the initial node n_I, which has in-degree zero, and the final node n_F, which has out-degree zero.
3. Every node of G occurs on some $n_I - n_F$ walk.

Here, the nodes of a control flow graph represent simple program statements and also branch conditions, while the edges represent possible transfers of control between these. The programs entry point and exit point are represented by the initial vertex and final vertex, respectively. A node of out-degree two in a control flow graph is called a *decision node*, and an edge incident from a *decision node* is called a *decision edge*. A *decision node* represents the branch condition of a conditional branch statement. If u is a *decision node* and $u_1, u_2 \in SUCC(u)$ are the successors of u. Then u_1 is called the *complement* of u_2 or vice-versa, with respect to u.

```
scanf(%d, &n);
sum=0;
while(n>0)
{
  sum=sum+1;
  n=n-1;
}
printf(%d, sum);
```

FIGURE 2.10: Program of sum of first n natural numbers

Example 2.8. *Figure 2.11 is the control flow graph of the program in Figure 2.10.*

Definition 2.19. *(Forward dominance)* Let G be a control flow graph. A node $n' \in N$ forward dominates a node $n \in N$ if every $n - n_F$ walk in G contains n'.

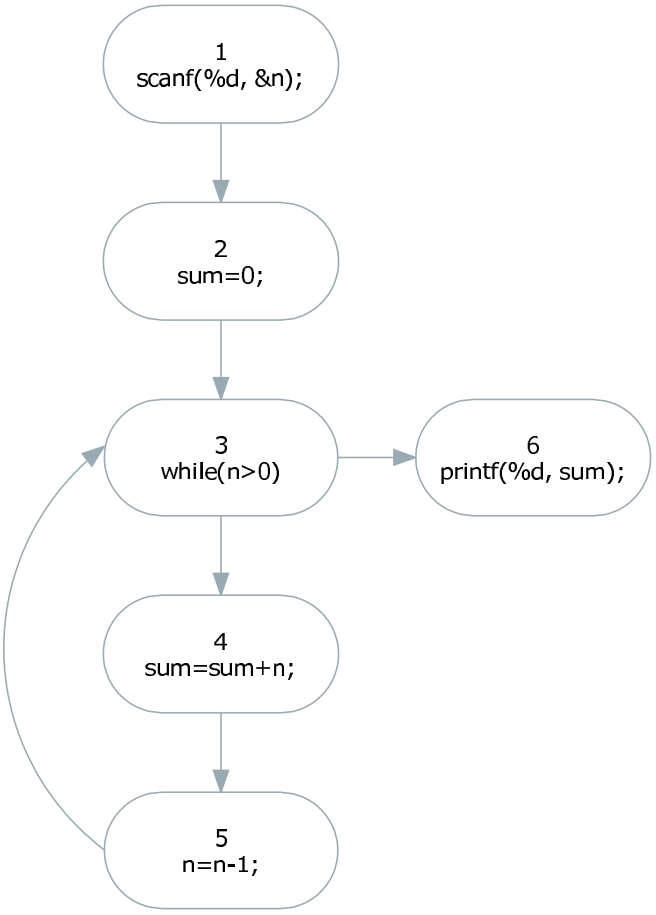


FIGURE 2.11: Control flow graph of Figure 2.10

n' *properly forward dominates* n iff $n' \neq n$ and n' forward dominates n. The *immediate forward dominator* of a node $n \in (N - \{n_F\})$ is the node that is the first proper forward dominator of n to occur on every $n - n_F$ walk in G. n' *strongly forward dominates* n iff n' forward dominates n and there is an integer $k \geq 1$ such that every walk in G beginning with n and of length $\geq k$ contains n'.

Example 2.9. *In the control flow graph of Figure 2.11 node 5 strongly forward dominates node 4, because there are arbitrarily long walks from node 4 that do not contain node 6. Whereas, node 6 is the immediate forward dominator of node 3.*

Definition 2.20. *(def/use graph)* A def/use graph is a quadruple $\mathcal{G} = (G, V, def, use)$, where G is a control flow graph, V is a finite set of symbols called variables, and $def : N \to \wp(V)$, $use : N \to \wp(V)$ are functions.

For each node $n \in N$, $def(n)$ represents the set of variables defined and $use(n)$ represents the set of variables used (referenced) at the statement represented by n, respectively. let W be a walk in G. Then, $def(W) = \bigcup_{n \in W} def(n)$.

Example 2.10. *Figure 2.12is the def/use graph of Figure 2.11.*

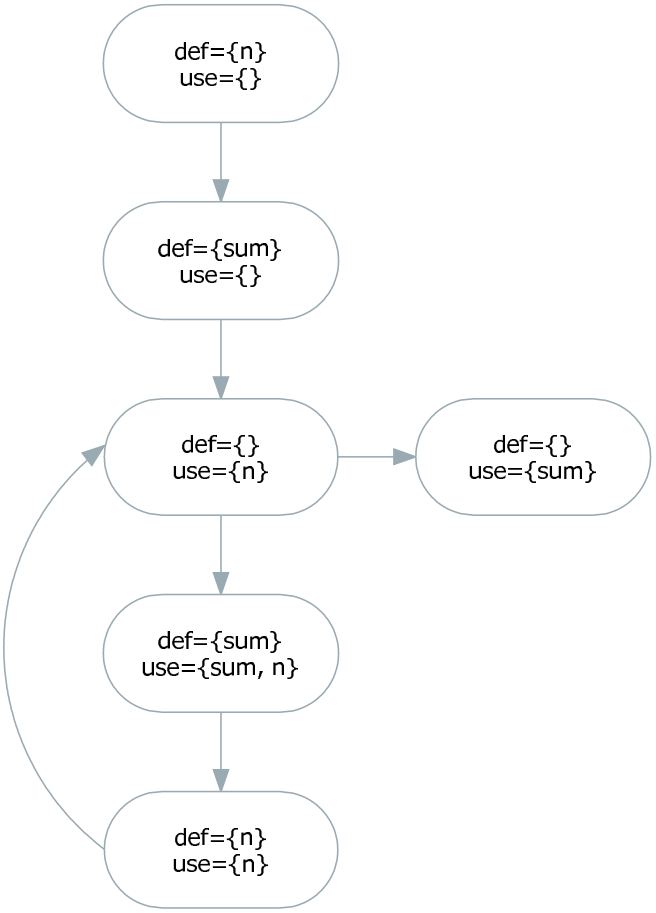


FIGURE 2.12: def/use graph of Figure 2.11

Definition 2.21. *(Data flow dependence)* Let $\mathcal{G} = (G, V, def, use)$ be a def/use graph, and let $n, n' \in N$. n' is *directly data flow dependent* on n if there is a walk w $(n-n')$ in G such that $(def(n) \cap use(n')) - def(W) \neq 0$. n' is *data flow dependent* on n if there is a sequence $n_1, n_2, ..., n_k$ of nodes, $n \geq 2$, such that $n = n_1$, $n' = n_k$, and n_i is directly data flow dependent on n_{i+1} for $i = 1, 2, ..., k-1$.

Definition 2.22. *(Strongly control dependent)* Let G be a control frow graph, and let $n, n' \in N$. n' is *strongly control dependent* on n iff there is a walk w $(n - n')$ in G not containing the immediate forward dominator of n.

Definition 2.23. *(Weakly control dependent)* Let G be a control frow graph, and let $n, n' \in N$. n' is *directly weakly control dependent* on n iff n has successors n'' and n''' such that n' *strongly forward dominates* n'' but does not strongly forward dominate n'''. n' is *weakly control dependent* on n iff there is a sequence $n_1, n_2, ..., n_k$ of nodes, $k \geq 2$, such that $n' = n_1$, $n = n_k$, and n_i is *directly weakly control dependent* on n_{i+1} for $i = 1, 2, .., k-1$.

Example 2.11. *Node 6 is (directly) weakly control dependent on node 3 (because node 6 strongly forward dominates itself, but not node 4), but not strongly control dependent on node 3 (because node 6 is the immediate forward dominator of node 3). In addition, node 3, node 4, and node 5 are (directly) weakly control dependent on node 3, because each strongly forward dominates node 4 but not node 6.*

The essential difference between weak and strong control dependence is that weak control dependence reflects a dependence between an exit condition of a loop and a statement outside the loop that may be executed after the loop is exited, while strong control dependence does not. Weak control dependence [21] is a generalization of strong control dependence in the sense that every strong control dependence is also a weak control dependence.

Definition 2.24. *(Weakly syntactically dependent)* Let $\mathcal{G} = (G, V, def, use)$ be a def/use graph, and let $n, n' \in N$. n' is *weakly syntactically dependent* on n if there is there is a sequence $n_1, n_2, ..., n_k$ of nodes, $k \geq 2$, such that $n' = n_1$, $n = n_k$, and for $i = 1, 2, .., k-1$ either n_i is *weakly control dependent* on n_{i+1} or ; n_i is *data flow dependent* on n_{i+l}.

Definition 2.25. *(Strongly syntactically dependent)* Let $\mathcal{G} = (G, V, def, use)$ be a def/use graph, and let $n, n' \in N$. n' is *strongly syntactically dependent* on n if there is there is a sequence $n_1, n_2, ..., n_k$ of nodes, $k \geq 2$, such that $n' = n_1$, $n = n_k$, and for $i = 1, 2, .., k-1$ either n_i is *strongly control dependent* on n_{i+1} or ; n_i is *data flow dependent* on n_{i+l}.

Example 2.12. *Node 6 is weakly syntactically dependent on node 5, because node 6 is weakly control dependent on node 3 and node 3 is data flow dependent on node 5; node 5 is strongly syntactically dependent on node 1, because node 5 is strongly control dependent on node 3 and node 3 is data flow dependent on node 1. Note that node 6 is not strongly syntactically dependent on v5.*

This notion of dependencies described so far loses some information, because syntactic occurrence is not enough to get the real idea of relevancy. For instance, the value assigned to x does not depend on y in the statement $x = z + y - y$, although y occurs in the expression. The syntactic approach may fail in computing the optimal set of dependencies, since it is not able to rule out this kind of *false dependencies*. This results in obtaining a slice which contains more statements than needed.The first step towards a generalization of the way of defining slicing

is to consider *semantic dependencies*, where intuitively a variable is relevant for an expression if it is relevant for its evaluation.

Definition 2.26. *(Semantic dependency)* Let $x, y \in \mathsf{Var}$, then the semantic dependency between the expression e and variable x is defined formally as,

$$\exists \sigma_1, \sigma_2 \in \Sigma. \forall y \neq x. \sigma_1(y) = \sigma_2(y) \wedge \mathcal{E}[\![e]\!]\sigma_1 \neq \mathcal{E}[\![e]\!]\sigma_2.$$

This semantic notion can then easily generalized in what we will call *abstract dependency*, where a variable is relevant to an expression if it affects a given property of its evaluation. More precisely, This notion of dependency is parametric on the properties of interest. Basically, an expression e depends on a variable x w.r.t. a property ρ if changing x, and keeping all other variables unchanged with respect to ρ, may lead to a change in e with respect to ρ.

Definition 2.27. *(Abstract dependency)* Let $x, y \in \mathsf{Var}$, then the abstract dependency between the expression e and variable x with respect to an abstract domain ρ (property) is defined formally as,

$$\exists \varphi_1, \varphi_2 \in \Sigma^{\rho}. \forall y \neq x. \varphi_1(y) = \varphi_2(y) \wedge \mathcal{H}[\![e]\!]\varphi_1 \neq \mathcal{H}[\![e]\!]\varphi_2.$$

2.4 Zero-knowledge Proofs

The fundamental notion of zero-knowledge was introduced by Goldwasser, Micali and Rackoff in [63]. They consider a setting where a powerful prover is proving a theorem to a probabilistic polynomial time verifier. Intuitively, a proof system is considered zero knowledge if whatever the verifier can compute while interacting with the prover it can compute by itself without going through the protocol. Informally, A zero-knowledge proof (ZKP) is a method by which one can prove knowledge of a fact without revealing that knowledge to another party. The word "proof" here is not used in the traditional mathematical since, rather, a "proof", or equivalently a "proof" system, is a randomized protocol by which one party (the prover) wishes to convince another party (the verifier) that a given statement is true.ZKPs exist only if one-way functions exist, as a cheating verifier may be able to extract additional information after interacting with a prover by essentially hacking the prover. The notion of one-way functions is often generalized to represent any means of committing to a secret bit (i.e., information hiding).

Example 2.13. *Let us consider the following example was adapted from [105].Peggy has found a magical cave(Figure 2.13). The cave has a magic door deep inside it that opens only upon uttering the secret word, a secret which Peggy has uncovered. Victor hears about this and wishes to also know the secret. Peggy agrees to sell Victor the secret word for $1,000,000, but Victor wants to be certain that Peggy, indeed, knows the secret word before he pays. How can Peggy (the prover) prove to Victor (the verifier) that she knows the secret without actually conveying the secret to Victor? Peggy and Victor devise the following scheme. Victor will wait outside the cave while Peggy enters. She chooses either path A or path B at random. Victor does not know which path she has chosen. Then, Victor will enter the cave as far as the fork and announce the path along which he wants Peggy to return.*

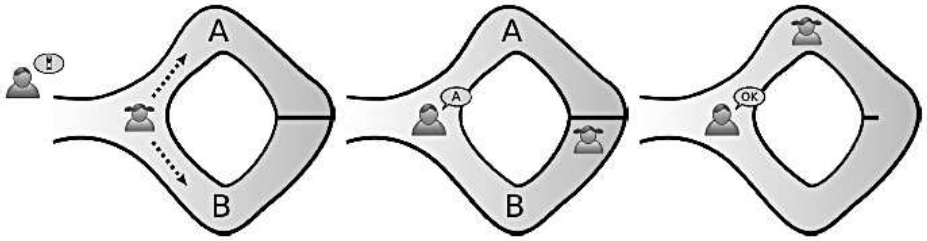


FIGURE 2.13: Magic Cave

Suppose Peggy knows the secret word. Then, she will be able to return along either path A or path B regardless of the path she chose initially. If Victor announces the same path through which Peggy chose to enter, she simply exits the cave along that path. If Victor announces the path that Peggy did not choose, she whispers the secret word (Victor is presumably too far away to hear it), thus opening the door and allowing her to return along the desired path. Suppose Peggy does not know the secret word. Then, she will only be able to return along the appropriate path if Victor announces the same path that she chose. This will occur with probability $\frac{1}{2}$*. If Peggy is tested many times, the probability that the path announced by Victor is the same chosen by Peggy for every test becomes negligible. That is, Victor will eventually discover that Peggy is a liar.*

One might wonder why Victor simply does not tell Peggy to enter through a known path (say, path A) and then require her to return along the other path. Clearly, this would force Peggy to use her knowledge of the secret word to return appropriately. However, such a scheme also allows Victor to eavesdrop by following her down the pre-specified path. By randomizing the initial path, the probability that Victor can successfully eavesdrop is reduced.

Chapter 3

Property Driven Program Slicing

Slicing enables large programs to be decomposed into ones which are smaller and hence potentially easier to analyse, maintain and test. A slice P_s of a program P w.r.t. a slicing criterion C has to

- be syntactically correct and executable; and
- give the same result as P if observations are restricted to C.

A slice is usually computed by analyzing how the effects of a computation are propagated through the code, i.e., by inferring dependencies. Following the theory of abstract interpretation [42] [43] [39], properties are abstractions of data. A general notion based on the observation that, in typical debugging tasks, the interest is often on the part of the program which is relevant to some property of data, rather than their exact value. studying dependence only as regards the abstraction, differently from the standard, concrete approach. In this chapter we redefine the traditional slicing algorithm [13, 36] by a data flow analysis which is based on the theory of Abstract Interpretation that enables an automatic extraction of information about all possible program executions, followed by a backward static slicing using the extracted information at each program point.

In Section 3.1, we categorize different forms of program slices. Mark Weiser's slicing algorthim is discussed in Section 3.2. Section 3.3 introduces the abstract state trajectories related to property driven program slicing. Dataflow based property driven program slicing algorithm is proposed in Section 3.4. In Section 3.5, we state the correctness conditions. Section 3.6 states the most relevant existing

program slicing techniques. In Section 3.7, we conclude by discussing the main advantages of our scheme.

3.1 Different Forms of Slice

Program slicing can be categorized in many forms like, backward or forward [129][107][58], static or dynamic [57][3][64], intra-procedural or inter-procedural [76]. Slicing has been applied to programs with arbitrary control flow (goto statements)[25][2] and even concurrent programming languages like Ada [24]. Another general form of slicing, Amorphous slicing [20] is a form of partial evaluation [21]. There are variants of slicing in between the two extremes of static and dynamic. Where some but not all properties of the initial state are known. These are known as conditioned slices [22] or constrained slices.[55]

3.1.1 Backward vs. Forward

Backward slicing answer the question "Which statements affect the slicing criterion?" This is the conventional one [129][130][132] where as forward slicing [107] is the reverse of backword slice, answers the question "Given a particular statement in a program. which other statements are affected by this particular statement's execution?"

3.1.2 Static vs Dynamic

A static slice is the conventional one where the slice required to agree with the program being sliced in all initial states. Dynamic slicing [26, 57, 84] involves executing the program in a particular initial state and using trace information to construct a slice to particular initial state.

3.1.3 Intra-procedural vs Inter-procedural

Intra-procedural slicing means slicing programs which do not have procedures whereas inter-procedural [76][107] slicing tackles the more complex problem of slicing programs where procedure definitions and calls are allowed.

3.1.4 Slicing Structured vs. Unstructured Programs

For many applications, particularly where the maintenance problems are the primary motivation for slicing, the slicing algorithm must be capable of constructing slices from unstructured programs as well. The archetype of this unstructured programming style is the go to statement, all forms of jump statement. Such as break and continue can be regarded as special cases of the goto statement. Such programs are said to exhibit arbitrary control flow and are considered to be unstructured. The traditional program dependence graph approach [100] incorrectly fails to include any goto statements in a slice. Various authors have suggested solutions to the problem [100][25][7].

3.1.5 Dataflow vs. Non-Dataflow

In the non-dataflow analysis approach [121], infeasible paths are detected by semantic dependency [91]. Parametric program slicing [55] is a non-dataflow approach of program slicing, where slices are constructed using a term rewriting system. which can use arbitrary rewrites which preserve a property of syntax using origin tracking.

Weiser's original work described backward, static, intra-procedural slicing although he also gave an algorithm for backward, static, inter-procedural slicing.

3.2 Weiser's Work

3.2.1 Dataflow Based Program Slicing

Weiser's program slicing algorithm is an example of data flow analysis. Data flow analysis [72, 130], by definition, is a act of inferring properties about program from its control flow graph(CFG) alone. The slicing algorithm takes the control flow graph G of a program P as input and outputs a set of nodes, $N_s \in N$. Since in this case there is one to one correspondence ($\sim$) between the nodes of the control flow graph and the statements of the corresponding program, this output uniquely determines which statements of P should be included in the slice of P. The slice

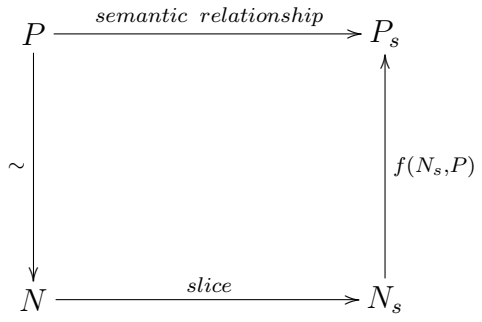


FIGURE 3.1: Weiser's slicing

P_s of P is the program derived from P and the set of nodes N_s output by Wiser's algorithm.

Let us illustrate the above idea pictorially (Figure 3.1). Weiser defines a slice as an executable program that is obtained from the original program by deleting zero or more statements. A slicing criterion C consists of a pair (n, V) where n is a node in the control flow graph (CFG) of the program, and V a subset of the programs variables present in n. In order to be a slice with respect to given criterion (n, V), a subset P_s of the statements of program P must satisfy the following properties:

1. P_s must be a valid program,
2. whenever P halts for a given input, P_s also halts for that input, computing the same values for the variables in V whenever the statement corresponding to node n is executed. At least one slice exists for any criterion: the program itself.

In Weiser's original thesis [129], it is shown how slices can be computed by solving a set of dataflow and control flow equations derived directly from the control flow graph (CFG) being sliced. These equations are solved using iterative process which entails computing sets of *relevant variables* for each node in the CFG from which sets of *relevant statements* are derived; the computed slice is defined as the fixpoint of the latter set.

Directly relevant variables

First, the *directly relevant variables* of node i, $R_C^0(i)$ are inductively defined as follows:

1. The set of directly relevant variables at slice node, n, is simply the slice set, V.
2. The set of directly relevant variables at every other node i, is defined in terms of the set of directly relevant variables of all nodes j leading directly from i to j ($i \longrightarrow_{CFG} j$) in the CFG. $R_C^0(i)$ contains all variables v such that either
   - $v \in R_C^0(j) - def(i)$ or
   - $v \in ref(i)$ and $def(i) \cap R_C^0(j) \neq \emptyset$

The directly relevant variables of a a node are the set of variables at that node upon which the slicing criterion is *transitively data dependent*.

Directly relevant statements

In terms of the directly relevant variables, a set of *directly relevant statements* S_C^0 is defined:

$$S_C^0 = \{i \mid \exists j \wedge (i \longrightarrow_{CFG} j) \wedge (def(i) \cap R_C^0(j) \neq \emptyset\})$$

Indirectly relevant variables

The subsequent iterations of Weiser's algorithm calculate the *indirectly relevant variables*, R_C^K where $K \geq 0$. In calculating the indirectly relevant variables, control dependency is taken into account.

$$R_C^{K+1}(i) = R_C^K(i) \cup \bigcup_{b \in B_C^K} R^0_{(b,use(b))}(i)$$

where

$$B_C^K = \{b \mid \exists i \in S_C^K \wedge (b \rightsquigarrow i\})$$

B_C^K is the set of all predicate nodes that *control* ($\rightsquigarrow$) a statement in S_C^K.

Indirectly relevant statements

Adding predicate nodes to S_C^K includes further indirectly relevant statements in the slice:

$$S_C^{K+1} = B_C^K \cup \{i \mid \exists j \wedge i \longrightarrow_{CFG} j \wedge def(i) \cap R_C^{K+1}(j) \neq \emptyset\}$$

This process will eventually terminate since S_C^K and R_C^K are non-decreasing subsets of program's variables. Weiser proves, in his thesis (Theorem 10), that his algorithm produces slices according to his semantic definition of a slice.

Example 3.1. *Let us consider a program which computes the sum and product of first n numbers, using a single loop. Figure 3.2 shows the program and its slice with respect to the slicing criteria* $C = (11, sum)$.

```
1. scanf("%d", &n);
2. if (n > 0) {
3.  i = 1;
4.  sum = 0;
5.  prod = 1;
6. while (i <= n) {
7.      sum = sum + i;
8.      prod = prod * i;
9.      i := i + 1;}
10. printf("%d",product);
11. printf("%d",sum);}
```

```
1. scanf("%d", &n);
2. if (n > 0) {
3.  i = 1;
4.  sum = 0;
5.
6. while (i <= n) {
7.      sum = sum + i;
8.
9.      i := i + 1;}
10.
11. printf("%d",sum);}
```

FIGURE 3.2: The original code and sliced code w.r.t C= (11, sum)

Table 3.1 and Table 3.2 show the computational steps of Weiser's slicing algorithm.

Now let us discuss how directly and indirectly variables and statements are computed using Weiser's algorithm on the above program consulting Table 3.1 and Table 3.2. Directly relevant statements S_C^0 *with slicing criteria* $C = (11, sum)$ *is computed from Table 3.1 using directly relevant variables* $R^0_{(11,sum)} = sum, i$ *as,* $S^0_{(11,sum)} = \{3, 4, 7, 11\}$. *The set of all predicate nodes in the program,* $B^0_{(11,sum)} =$

No	statement	def	use	$R^0_{(11,sum)}$
1	scanf("%d", &n);	{n}	∅	∅
2	if (n > 0) {	∅	{n}	∅
3	i = 1;	{i}	∅	∅
4	sum = 0;	{sum}	∅	{i}
5	prod = 1;	{prod}	∅	{sum, i}
6	while (i <= n) {	∅	{i, n}	{sum, i}
7	sum = sum + i;	{sum}	{sum, i}	{sum, i}
8	prod = prod * i;	{prod}	{prod, i}	{sum}
9	i = i + 1;}	{i}	{i}	{sum}
10	printf("%d", prod);	∅	{prod}	{sum}
11	printf("%d", sum);}	∅	{sum}	{sum}

TABLE 3.1: Computing $R^0_{(11,sum)}$

No	statement	def	use	$R^0_{(2,n)}$	$R^0_{(6,n)}$	$R^0_{(6,i)}$
1	scanf("%d", &n);	{n}	∅	∅	∅	∅
2	if (n > 0) {	∅	{n}	{n}	{n}	∅
3	i = 1;	{i}	∅	∅	{n}	∅
4	sum = 0;	{sum}	∅	∅	{n}	{i}
5	prod = 1;	{prod}	∅	∅	{n}	{i}
6	while (i <= n) {	∅	{i, n}	∅	{n}	{i}
7	sum = sum + i;	{sum}	{sum, i}	∅	{n}	{i}
8	prod = prod * i;	{prod}	{prod, i}	∅	{n}	{i}
9	i = i + 1;}	{i}	{i}	∅	{n}	{i}
10	printf("%d", prod);	∅	{prod}	∅	{n}	{i}
11	printf("%d", sum);}	∅	{sum}	∅	{n}	{i}

TABLE 3.2: Computing $\bigcup_{b \in B^0_{(11,sum)}} R^0_{(b,use(b))}$

$\{2, 6\}$. *The indirectly relevant variables are computed in Table 3.2, Indirectly relevant variables are* $R^1_{(11,sum)} = \{sum, i, n\}$. *Indirectly relevant statements are calculated as* $S^1_{(11,sum)} = \{2, 6\} \cup \{3, 4, 7\} \cup \{1, 9\} = \{1, 2, 3, 4, 6, 7, 9, 11\}$.

3.2.2 Weiser's Semantics Definition of Valid Slices

The essential issue in program slicing is to define what semantic relationship must exist between a program and its slice in order that the slice is considered valid. Mark Weiser [129] defined the semantic relationship that must exist between a program and its slice in terms of state trajectories.

Definition 3.1. *(Trajectories)* A state trajectory is a finite sequence of ordered pairs:

$$\tau = \langle (p_0, \sigma_0), (p_1, \sigma_1), ..., (p_k, \sigma_k) \rangle$$

where $p_0, p_1, ..., p_k$ are the program points in program P and $\langle p_0, p_1, ..., p_k \rangle$ is the path to be traversed during program execution. $\forall \sigma_i \in \Sigma$, be a state which is assumed immediately before execution of p_i. More formally, $\sigma \in \Sigma : V \rightarrow \mathcal{V}$ are memory configurations. i.e., mappings from variables V to values $\mathcal{V}$ ($\mathcal{V} \stackrel{\text{def}}{=} \mathbb{Z}$) .

3.2.3 Trajectory Semantics

Let us revise the trajectory semantics [9][129] on a simple statement oriented imperative programming language WHILE by denotational semantics [99]. In standard denotational semantics, states $\sigma \in \Sigma : V \rightarrow \mathcal{V}$ are memory configurations. i.e., mappings from variables V to values $\mathcal{V}$ ($\mathcal{V} \stackrel{\text{def}}{=} \mathbb{Z}$). The abstract syntax of the language is given by Table 3.3. Standard trajectory semantics, $\tau[\![.]\!]$, is a map $S \rightarrow \{L \times \Sigma\}$. We now give rules which define τ for each syntactic category:

Value domains	
$x, y \in \mathsf{Var}$	variables
$n \in \mathsf{Num}$	numerals
$l \in \mathsf{Lab}$	labels
Syntactic categories	
$a \in \mathsf{AExp}$	arithmetic expressions
$b \in \mathsf{BExp}$	boolean expressions
$s \in \mathsf{Stmt}$	statements
Operators	
$op_a \in \mathsf{Op}_a$	arithmetic operators
$op_b \in \mathsf{Op}_b$	boolean operators
$op_r \in \mathsf{Op}_r$	relational operators
Abstract syntax	
$a ::= x \mid n \mid a_1\ op_a\ a_2$	
$b ::= true \mid flase \mid not\ b \mid b_1\ op_a\ b_2 \mid$	
$\quad b_1\ op_r\ b_2 \mid a_1\ op_r\ a_2$	
$S ::= l : skip \mid l : x := a \mid S_1; S_2 \mid$	
$\quad l : if\ b\ then\ S_1\ else\ S_2 \mid$	
$\quad l : while\ b\ do\ S$	

TABLE 3.3: abstract syntax of WHILE

- For *skip* statement:

$$\boxed{\tau[\![l: \ skip]\!]\sigma = \langle(l,\sigma)\rangle}$$

$\langle(l,\sigma)\rangle$ represents the singleton sequence consisting of the pair (l,σ).

- For assignment statement:

$$\boxed{\tau[\![l: \ x = a]\!]\sigma = (l, \sigma[x \leftarrow \mathcal{E}[\![a]\!]\sigma])}$$

where $\mathcal{E}[\![a]\!]\sigma$ means the *new* value resulting from evaluating expression a.

- For sequences of statements:

$$\boxed{\tau[\![l: \ S_1; S_2]\!]\sigma = \tau[\![S_1]\!]\sigma \ \diamond \ \tau[\![S_2]\!]\sigma'}$$

where σ' is the state obtained after executing S_1 in σ and $\diamond$ means concatenation.

- For *if* statement:

$$\boxed{\tau[\![l: \ if \ b \ then \ S_1 \ else \ S_2]\!]\sigma = \langle(l,\sigma)\rangle \ \diamond (\mathcal{E}[\![b]\!]\sigma \rightarrow \tau[\![S_1]\!]\sigma, \tau[\![S_2]\!]\sigma)}$$

The first element of the trajectory is the label of the *if* in the current state. The rest of the trajectory is the trajectory of one of the branches depending on the value of the boolean expression evaluated in the current state.

- For *while* statement:

$$\boxed{\tau[\![l: \ while \ b \ then \ S]\!]\sigma = \tau[\![if(l: \ b) \ \{S; while(l: \ b) \ S\} else \ skip]\!]\sigma}$$

while loops are defined simply in terms of *if* statements in the standard way.

Since a slice only needs to preserve the behavior of the program with respect to the variables of interest, the concept of state restriction is introduced.

Definition 3.2. *(Restriction of a state to a set of variables)* Given a state, σ and a set of variables V, $\sigma|_V$ restricts σ so that it is defined only for variables in V:

$$(\sigma|_V)x = \begin{cases} \sigma x & if \;\; x \in V \\ \bot & otherwise \end{cases}$$

Since the programs only need to agree at the slicing criterion, the idea of restricting a trajectory to a slicing criterion (p, V) is introduced. To do this, first delete all pairs in the trajectory whose label component is not p and for the ones that are left, restrict the state component of the pair to V as just defined. First we define how to project a single element of a trajectory onto a slicing criterion:

Definition 3.3. *(Projection of a trajectory to a slicing criterion)* For a program point p' and a state σ the projection of the trajectory sequence element (p', σ) to the slicing criterion (p, V) is

$$(p', \sigma)|_{(p,V)} = \begin{cases} (p', \sigma|_V) & if \;\; p' = p \\ \lambda & otherwise \end{cases}$$

where λ denotes the empty string.

The projection of the trajectory $\tau = \langle (p_0, \sigma_0), (p_1, \sigma_1), ..., (p_k, \sigma_k) \rangle$ to the slicing criterion (p, V) is

$$Proj_{(p,V)}(\tau) = \langle (p_0, \sigma_0)|_{(p,V)}, (p_1, \sigma_1)|_{(p,V)}, ..., (p_k, \sigma_k)|_{(p,V)} \rangle$$

Definition 3.4. *(Weiser's backward static slice)* A slice P' of a program P on a slicing criterion (p, V) is any executable program with the following two properties:

1. P' can be obtained from P by deleting zero or more statements.
2. Whenever P halts on an input state σ with a trajectory τ then P' also halts on input σ with trajectory τ' where $Proj_{(p,V)}(\tau) = Proj_{(p,V)}(\tau')$.

The trajectory $Proj_{(p,V)}(\tau)$ is obtained first by deleting all elements of τ whose label component is not p and then by restricting the state components to V.

$$\mathcal{H}[\![a]\!]\varphi = \begin{cases} d_i & if \;\; a = x_i \in \mathsf{Var} \\ \alpha(n) & if \;\; a = n \in \mathsf{Num} \\ \mathcal{H}[\![a_1]\!]\varphi \;\; \widehat{op_a} \;\; \mathcal{H}[\![a_2]\!]\varphi & if \;\; a = a_1 \; op_a \; a_2 \end{cases}$$

TABLE 3.4: Approximation of arithmetic expressions

3.3 Abstract Semantics

We consider the WHILE language in Table 3.3. The set of concrete states Σ consists of functions $\sigma : \mathsf{V} \to \mathcal{V}$ which maps the variables to their values from the semantic domain $\mathbb{Z}_\perp$ where, $\perp$ represents an undefined or uninitialized value and $\mathbb{Z}$ is the set of integers. If a program has k variables $x_1, ..., x_k$, we can represent states as tuples, i.e., $\sigma = \langle x_1, ..., x_k \rangle$ and $\Sigma = \mathcal{V}^k$.

The semantics of arithmetic expression $a \in \mathsf{AExp}$ over the state σ is denoted by $\mathcal{E}[\![a]\!]\sigma$ where, the function $\mathcal{E}$ is of the type $\mathsf{AExp} \to (\sigma \to \mathcal{V})$. Similarly, $\mathcal{B}[\![b]\!]\sigma$ denotes the semantics of boolean expression $b \in \mathsf{BExp}$ over the state σ of type $\mathsf{BExp} \to (\sigma \to T)$ where T is the set of truth values.

Let $\mathcal{D}$ be an abstract domain on concrete values and α and γ are *abstraction* and *concretization* functions, respectively. The related abstract semantics on expressions , $\mathcal{H}[\![a]\!]\varphi$, is applied to abstract states $\varphi = \langle d_1, ..., d_k \rangle$ and $\varphi \in \mathcal{D}^k$ and is defined as the best correct approximation of $\mathcal{E}[\![a]\!]\sigma$ in Table 3.4, $\widehat{op_a}$ is the abstract operation in $\mathcal{D}$ that safely approximate op_a.

When we construct the approximate or abstract semantics of programs, we need to define abstract operations over the abstract domain, that approximate concrete operations over the concrete domain. The idea is that the abstract calculation *simulates* the concrete calculation, and the concretization of the abstract calculation is a correct approximation of the values in the concrete result.

Example 3.2. *For example consider the following code fragment in Figure 3.3 and consider the abstract domain where the addition and multiplication are influenced according to the well known rule of signs*

```
1. x=2;
2. y=-5;
3. z = (x+3)*y;
```

FIGURE 3.3: Sample code fragment

$$\mathcal{H}_b[\![b]\!]\varphi = \begin{cases} TRUE & if \;\; b = TRUE \\ & OR \\ & b = a_1 \; op_r \; a_2 \;\; AND \\ & \mathcal{H}[\![a_1]\!]\varphi \;\; \widehat{op_r} \;\; \mathcal{H}[\![a_2]\!]\varphi = TRUE \\ FALSE & if \;\; b = FALSE \\ & OR \\ & b = a_1 \; op_r \; a_2 \;\; AND \\ & \mathcal{H}[\![a_1]\!]\varphi \;\; \widehat{op_r} \;\; \mathcal{H}[\![a_2]\!]\varphi = FALSE \\ ? & undefined \; otherwise \end{cases}$$

TABLE 3.5: Approximation of boolean expressions

$$\begin{aligned} \mathcal{H}[\![x + 3 * y]\!]\varphi &= (\mathcal{H}[\![x]\!]\varphi \mathbin{\widehat{+}} \mathcal{H}[\![3]\!]\varphi) \mathbin{\overline{*}} \mathcal{H}[\![y]\!]\varphi \\ &= (+ \mathbin{\widehat{+}} \alpha(3)) \mathbin{\widehat{*}} - \\ &= (+ \mathbin{\widehat{+}} +) \mathbin{\widehat{*}} - \\ &= + \mathbin{\widehat{+}} - \\ &= - \end{aligned}$$

The abstract semantics $\mathcal{H}_b[\![b]\!]\varphi$ of boolean expression b is defined as the best correct approximation of $\mathcal{B}[\![b]\!]\sigma$ in Table 3.5, $\widehat{op_r} : \mathcal{D} \times \mathcal{D} \to \{TRUE, FALSE, ?\}$ is the abstract operation that safely approximate op_r and ? ($undefined$) signifies that, the abstract domain is not accurate enough to evaluate the condition.

Example 3.3. *Let* $\widehat{Op_r} = \{<, \leq, >, \geq, \neq, =\}$*, now consider the abstract operations* $\widehat{\leq}$ *and* $\widehat{\neq}$ *on Sign domain (Table 3.6), other relational operators can be abstracted accordingly.*

$\widehat{\neq}$	$\top$	$\bot$	$+$	0	$-$
$\top$	?	?	?	?	?
$\bot$	?	?	?	?	?
$+$	?	?	?	TRUE	TRUE
0	?	?	TRUE	?	TRUE
$-$	?	?	TRUE	TRUE	?

$\widehat{<}$	$\top$	$\bot$	$+$	0	$-$
$\top$	?	?	?	?	?
$\bot$	?	?	?	?	?
$+$	?	?	?	FALSE	FALSE
0	?	?	TRUE	?	FALSE
$-$	?	?	TRUE	TRUE	?

TABLE 3.6: Abstracting $\neq$ and $<$ operator

3.3.1 Abstract Trajectory

We now define the abstract trajectory semantics for WHILE in Table 3.3.

- For *skip* statement:

  $$\boxed{\tau^{\mathcal{D}}[\![l:\ skip]\!]\varphi = \langle(l,\varphi)\rangle}$$

  $\langle(l,\varphi)\rangle$ represents the singleton sequence consisting of the pair (l,φ). i.e statement level alone with the properties of the variables.

- For assignment statement:

  $$\boxed{\tau^{\mathcal{D}}[\![l:\ x=a]\!]\varphi = (l,\varphi[x \leftarrow \mathcal{H}[\![a]\!]\varphi])}$$

  where $\mathcal{H}[\![a]\!]\varphi$ means the *new* value resulting from evaluating expression a in abstract domain and $\varphi[x \leftarrow \mathcal{H}[\![a]\!]\varphi]$ is the abstract state φ *updated* with the maplet that takes variable x to this new abstract value.

- For sequences of statements:

  $$\boxed{\tau^{\mathcal{D}}[\![l:\ S_1;S_2]\!]\varphi = \tau[\![S_1]\!]\varphi\ \diamond\ \tau^{\mathcal{D}}[\![S_2]\!]\varphi'}$$

  where φ' is the abstract state obtained after executing S_1 in φ and $\diamond$ means concatenation.

☐ For *if* statement:

$$\tau^{\mathcal{D}}[\![l: \ if\ b\ then\ S_1\ else\ S_2]\!]\varphi = \langle (l,\varphi)\rangle \diamond$$

$$\begin{cases} \bot & if\ \ \mathcal{H}_b[\![b]\!]\varphi =? \\ \tau^{\mathcal{D}}[\![S_1]\!] & if\ \mathcal{H}_b[\![b]\!]\varphi = TRUE \\ \tau^{\mathcal{D}}[\![S_2]\!] & if\ \mathcal{H}_b[\![b]\!]\varphi = FALSE \\ (\tau^{\mathcal{D}}[\![S_1]\!]\varphi) \sqcup (\tau^{\mathcal{D}}[\![S_2]\!]\varphi) & otherwise \end{cases}$$

The first element is the label of the *if* in the current abstract state. The rest of the trajectory is the trajectory of one of the branches depending on the abstract execution of the boolean expression evaluated in the current abstract state.

☐ For *while* statement:

$$\tau^{\mathcal{D}}[\![l: \ while\ b\ then\ S]\!]\varphi =$$

$$\begin{cases} \lambda & if\ \mathcal{H}_b[\![b]\!]\varphi = FALSE \\ \langle l_i, \sqcup_{i\geq 0}(\varphi_i)\rangle & otherwise \end{cases}$$

If the predicate b evaluated to be *FALSE* there would be a empty trajectory at l other wise a fixpont iteration on the abstract state of each statements with in the loop body where $\varphi_0 = \varphi$ and $\varphi_{i+1} = \tau^{\mathcal{D}}[\![S]\!]\varphi_i$

Definition 3.5. *(Restriction of a state to a set of variables w.r.t a given property)* Given a abstract state, φ with respect to a property, ρ and a set of variables, $V \in \mathsf{Var}$, $\varphi|_V^\rho$ restricts φ so that it is defined by ρ only for variables in V.

Definition 3.6. *(Projection of a abstract trajectory to a slicing criterion w.r.t a given property)* For a program point p' and a abstract state φ, the projection of the abstract trajectory sequence element (p', φ) to the slicing criterion (p, V) w.r.t property ρ is

$$(p',\varphi)|_{(p,V)}^\rho = \begin{cases} (p', \varphi|_V^\rho) & if\ \ p' = p \\ \lambda & otherwise \end{cases}$$

where λ denotes the empty string.

begin
i=0;
while($i < n$){
  j=1;
  while($p_{i+j} = p_i$) && ($\varphi_{i+j} = \varphi_i$)
    remove (p_{i+j}, φ_j) from the trajectory
  i=i+j;
}

TABLE 3.7: $\mathscr{R}ed$

The projection of the abstract trajectory $\tau^{\mathcal{D}}$ to the slicing criterion (p, V) w.r.t a property ρ is

$$Proj_{(p,V)}(\tau^{\mathcal{D}}) = \langle (p_0, \varphi_0)|^{\rho}_{(p,V)}, (p_1, \varphi_1)|^{\rho}_{(p,V)}, ..., (p_k, \varphi_k)|^{\rho}_{(p,V)} \rangle$$

Definition 3.7. *(Property driven program slicing)* A property driven slice P_ρ of a program P on a slicing criterion (p, V) and with respect to a given property ρ is any executable program with the following two properties:

- ☐ P' can be obtained from P by deleting zero or more statements.
- ☐ Whenever P halts on an input state φ with a abstract trajectory $\tau^{\mathcal{D}}$ then P' also halts on φ with trajectory $\tau^{\mathcal{D}'}$ where,

$$\mathscr{R}ed(Proj_{(p,V)}(\tau^{\mathcal{D}})) = \mathscr{R}ed(Proj_{(p,V)}(\tau^{\mathcal{D}'})).$$

Where $\mathscr{R}ed$ is defined in Table 3.7, given a abstract trajectory $\tau^{\mathcal{D}} = \langle (p_0, \varphi_0), (p_1, \varphi_1), ..., (p_k$
$\mathscr{R}ed$ is obtained by applying the following reduction algorithm,

Example 3.4. *Consider Table 3.8 for an illustration of the above definitions,*

The abstract state trajectory of program P with respect to Sign property is denoted as τ^{Sign} and the abstract state trajectory of the sliced program P_{Sign} with respect to the property Sign on slicing criteria $C = (10, w)$ is denoted as $\tau^{Sign'}$.

$$\tau^{Sign} = \langle (1, \{\perp, \perp, \perp, \perp\}), (2, \{\perp, +, \perp, \perp\}), (3, \{\perp, +, +, \perp\}), (4, \{\perp, +, +, -\}),$$
$$(5, \{\perp, +, +, -\}), (6, \{-, +, +, -\}), (10, \{-, +, +, -\}) \rangle$$

St.No.	Original Program, P	Sliced Program, P_{Sign}
1	$x = 5;$	$x = 5;$
2	$y = 3;$	$y = 3;$
3	$z = y - x;$	$z = y - x;$
4	$if(x > z)\{$	$if(x > z)$
5	$y = x + z^2;$	
6	$w = y * z;\}$	$w = y * z;$
7	$else\{$	
8	$y = x^2 + z;$	
9	$w = y * z;\}$	
10	$printf"\%d", w;$	$printf"\%d", w;$

TABLE 3.8: Property driven slicing

$$\tau^{Sign'} = \langle (1, \{\bot, \bot, \bot, \bot\}), (2, \{\bot, +, \bot, \bot\}), (3, \{\bot, +, +, \bot\}), (4, \{\bot, +, +, -\}), (6, \{\bot, +, +, -\}), (10, \{-, +, +, -\})\rangle$$

Notice that,

$$\mathscr{R}ed(Proj_{(10,w)}(\tau^{Sign})) = \mathscr{R}ed(Proj_{(10,w)}(\tau^{Sign'}))$$

3.4 Dataflow Based Property Driven Program Slicing

The algorithm involve a data flow analysis which is based on the theory of Abstract Interpretation that enables an automatic extraction of information about all possible program executions, followed by a backward static slicing using the extracted information at each program point.

3.4.1 Phase 1: Static Analysis

We will restrict to boolean and integer valued variables, and will be interested in approximating functions over integers (such as addition, subtraction, multiplication and addition), i.e., functions of the type $f : \mathbb{Z}^n \to \mathbb{Z}$. However, the concrete domain used in abstract interpretation operates over sets of integers rather than integers themselves. Thus, for any $n - ary$ operation, $f : Z^n \to Z$, it is possible

to define a lifted version $f_P : \wp(Z)^n \to \wp(Z)$. In practice, when operations over the integers are used, the concrete domain will be $\wp(Z)$.

Our representation of programs are *def/use* graphs. The objective of a static analysis based on Abstract Interpretation is to assign sets of possible abstract values to edges of a *def/use* graph. The *def/use* graph consists of five different node types which represent program points:

1. A designated start and end node representing the beginning and end point of a *def/use* graph.
2. Expression nodes representing different expression types found in a concrete semantic model.
3. Condition nodes representing forks in a control flow, i.e. this type of nodes has one incoming and two outgoing edges.
4. Join nodes merging two paths of the *def/use* graph, i.e. these nodes have two incoming and one outgoing edge.

Like the classical approach, our analysis also begins at the start node of the *def/use* graph and traverses the graph during its static program analysis phase. Depending on the encountered node type, a particular set of rules which is based on Abstract Interpretation is applied.

Based on the *def/use* graph, the classical approach begins with the construction of a complete transition system for the five node types. It defines how an abstract state is transferred into one state to another state at program point p:

$$\mathscr{T}_p : \wp(\Sigma^{\mathcal{A}}) \to \wp(\Sigma^{\mathcal{A}})$$

The transition system $\mathscr{T}$ is used to construct a system of equations which define the assignment of abstract states to program points. A solution is found by a *fixed-point* iteration. It begins with the least possible assignment $\mathscr{T}(\bot)$ where $\bot$ is the least element representing $\emptyset$. The *fixed-point* iteration continues as long as a further application of $\mathscr{T}$ does not compute a new state: $\mathscr{T}^0 = \bot$ *and* $\mathscr{T}^{n-1} = \mathscr{T}^n$.

Now we will define $\mathscr{T}$ for the different types of control flow edges in a *def/use* graph. For any edge $\mathfrak{e} \in E$ we shall denote its predecessor edges as $\mathfrak{e}_{pre}$. For

merge nodes, which have two incoming edges, the second is denoted $\mathfrak{e}_{pre'}$. In the following, $\mathscr{T}$ is given for every type of program point with respect to a given abstract domain ρ. $\forall\varphi_\rho \in \Sigma^\rho$ denotes the abstract states associated to program variables at each program point.

Start edge: At the start edge $\mathfrak{e}$, nothing is known about the values of variables. Having said this, the natural definition of an abstract state associated with the initial state should be as follows:

$$\boxed{\mathscr{T}_{\mathfrak{e}}(\varphi_\rho) = \bot}$$

Assignment edge: An assignment edge is an edge which emerges from an assignment node. Let, an assignment node has an assignment $x := a$ associated with it, where $x \in \mathsf{Var}$ and $a \in \mathsf{AExp}$, then $\mathscr{T}_{\mathfrak{e}}(\varphi_\rho)$ should be equal to the previous abstract state with the variable x updated to the abstract value of e (Table 3.4), as follows:.

$$\boxed{\mathscr{T}_{\mathfrak{e}}(\varphi_\rho) = \mathscr{T}_{\mathfrak{e}_{pre}}(\varphi_\rho[x \leftarrow \mathcal{H}[\![a]\!]\varphi_\rho])}$$

Merge edge: The problem of Abstract Interpretation is that a termination of the fixed-point iteration can not be guaranteed. Due to the nature of Abstract Interpretation which iteratively simulates each state transition, the fixed-point iteration can consume a significant amount of time for loops with large iteration counts. To overcome both problems, the widening operator ∇[44][38] can be applied. Its application typically enlarges the abstract states during the fixed-point iteration leading to a correct but also over-approximated solution which might become infeasible as result for many applications. Thus, a narrowing operator $\triangle$ was introduced [44][38] to restrict the over-approximation afterwards.

A merge edge is an edge emerging from a merge node. A merge node combines the analysis results of the two incoming edges. The least abstract value which is correct with respect to both incoming values is the supremum of the these. In addition, if the merge node is the entry of a loop, then that is a good place to put the widening based on the abstract domain. Thus, the abstract transition function for merge nodes is

$$\mathscr{T}_{\mathfrak{e}}(\varphi_\rho) = \begin{cases} \mathscr{T}_{\mathfrak{e}}(\varphi_\rho) \nabla (\mathscr{T}_{\mathfrak{e}_{pre}}(\varphi_\rho) \sqcup \mathscr{T}_{\mathfrak{e}_{pre'}}(\varphi_\rho)) & if\ loop\ merge \\ \mathscr{T}_{\mathfrak{e}_{pre}}(\varphi_\rho) \sqcup \mathscr{T}_{\mathfrak{e}_{pre'}}(\varphi_\rho) & otherwise \end{cases}$$

Conditional edges: The conditional node has two outgoing edges. Conditionals are resolved by only boolean expressions with relational operators Op_r, so for an abstract domain it is necessary to have abstract version of all relational operators $\widehat{Op_r}$ (Table 3.5).

$$\mathscr{T}_{\mathfrak{e}}(\varphi_\rho) = \begin{cases} \mathscr{T}_{\mathfrak{e}_{pre}}(\varphi_\rho) \wedge \mathcal{H}_b[\![b]\!]\varphi_\rho = TRUE \\ \mathscr{T}_{\mathfrak{e}_{pre}}(\varphi_\rho) \wedge \mathcal{H}_b[\![b]\!]\varphi_\rho = FALSE \end{cases}$$

The abstract interpretation may establish certain properties of a program through which we can identify infeasible statements of the program which will not be taken into account for program execution by predicting predicates present in conditional statements. By the following rules we modify the program P in order to simplify the control dependence, taking into account only the statements that have all impact on the property of interest ρ. Here, $P[S'/S]$ represents the replacement of S by S' in P yields the simplified program P'.

Example 3.5. *Lets apply the rules in Table 3.9 on the following code fragments, In Table 3.10, P' is obtained by applying rule 1(a) on P by statically analyzing the program in Parity domain and in Table 3.11 we apply rule 2(a) on P analyzing the Sign property. In both cases P' contains less statements than P. But in Table 3.12 we notice no improvement in terms of the number of statements both in P and P'. Therefore, the above rules can often generate a reduced CFG by statically analyzing the associated program with respect to a certain property ρ.*

Rule 1 For S::= $l: if\ b\ then\ S_1\ else\ S_2$
$\begin{cases} (a) P' = P[S\ /\ S_1] & if\ \mathcal{H}_b[\![b]\!]\varphi_\rho = TRUE \\ (b) P' = P[S\ /\ S_2] & if\ \mathcal{H}_b[\![b]\!]\varphi_\rho = FALSE \\ (c) P' = P[S\ /\ S] & No\ replacement\ otherwise \end{cases}$
Rule 2 For S::= $l: while\ b\ do\ S_1$
$\begin{cases} (a) P' = P[skip\ /\ S] & if\ \mathcal{H}_b[\![b]\!]\varphi_\rho = FALSE \\ (b) P' = P[S\ /\ S] & No\ replacement\ otherwise \end{cases}$

TABLE 3.9: Rules for conditional nodes

P	$\varphi\{w,x,y,z\}$	P'	$\varphi\{w,x,y,z\}$
$scanf("\%d", \&z);$	$(\bot,\ \bot,\ \bot,\ \bot)$	$scanf("\%d", \&z);$	$(\bot,\ \bot,\ \bot,\ \bot)$
$y = 15;$	$(\bot,\ \bot,\ \bot,\ \top)$	$y = 15;$	$(\bot,\ \bot,\ \bot,\ \top)$
$x = 2 * z;$	$(\bot,\ \bot,\ O,\ \top)$	$x = 2 * z;$	$(\bot,\ \bot,\ O,\ \top))$
$if(x! = y)$	$(\bot,\ E,\ O,\ \top)$		
$\quad w = x + y;$	$(\bot, E,\ O,\ \top)$	$w = x + y;$	$(\bot, E,\ O,\ \top)$
$else$	$(\bot,\ E,\ O,\ \top)$		
$\quad w = x - y + 1;$	$(\bot,\ E,\ O,\ \top)$		
$printf("\%d", w);$	$(\top,\ E,\ O,\ \top)$	$printf("\%d", w);$	$(O, E,\ O,\ \top)$

TABLE 3.10: Application of rule 1(a) on program P

P(Original program)	$\varphi\{i,j,x,y\}$	P′(applying Rule 4)	$\varphi\{i,j,x,y\}$
$i = 1;$	$(\bot,\ \bot,\ \bot,\ \bot)$	$i = 1;$	$(\bot,\ \bot,\ \bot,\ \bot)$
$y = 2;$	$(+,\ \bot,\ \bot,\ \bot)$	$y = 5;$	$(+,\ \bot,\ \bot,\ \bot)$
$x = 1;$	$(+,\ \bot,\ \bot,\ +)$	$x = 3;$	$(+,\ \bot,\ \bot,\ +)$
$j = 5 * (x - y);$	$(+,\ \bot,\ +,\ +)$	$j = 5 * (x - y);$	$(+,\ \bot,\ +,\ +)$
$while(i < j)\{$	$(+,\ -,\ +,\ +)$		
$\quad x = x + y;$	$(+,\ -,\ +,\ +)$		
$\quad i = i + 1;\}$	$(+,\ -,\ +,\ +)$		
$printf("\%d", x);$	$(+,\ -,\ +,\ +)$	$printf("\%d", x);$	$(+,\ -,\ +,\ +)$

TABLE 3.11: Application of rule 2(a) on program P

As an illustration consider CFG (for simplicity and better illustration we consider control flow graphs since def/use graph is a augmented version of CFG so this technique can be easily adopted to def/use graphs as well) in Figure 3.4. If the abstract domain of signs is used then the abstract transition function yields the following system of equations in Table 3.13.

P(Original program)	$\varphi\{i,j,x,y\}$	P′(applying Rule 4)	$\varphi\{i,j,x,y\}$
$i=1;$	$(\perp,\ \perp,\ \perp,\ \perp)$	$i=1;$	$(\perp,\ \perp,\ \perp,\ \perp)$
$y=2;$	$(+,\ \perp,\ \perp,\ \perp)$	$y=5;$	$(+,\ \perp,\ \perp,\ \perp)$
$x=1;$	$(+,\ \perp,\ \perp,\ +)$	$x=3;$	$(+,\ \perp,\ \perp,\ +)$
$j=5*(y-x);$	$(+,\ \perp,\ +,\ +)$	$j=5*(x-y);$	$(+,\ \perp,\ +,\ +)$
$while(i<j)\{$	$(+,\ +,\ +,\ +)$	$while(i<j)\{$	$(+,\ +,\ +,\ +)$
$x=x+y;$	$(+,\ +,\ +,\ +)$	$x=x+y;$	$(+,\ +,\ +,\ +)$
i=i+1;}	$(+,\ +,\ +,\ +)$	i=i+1;}	$(+,\ +,\ +,\ +)$
$printf("\%d",x);$	$(+,\ +,\ +,\ +)$	$printf("\%d",x);$	$(+,\ +,\ +,\ +)$

TABLE 3.12: Application of rule 2(b) on program P

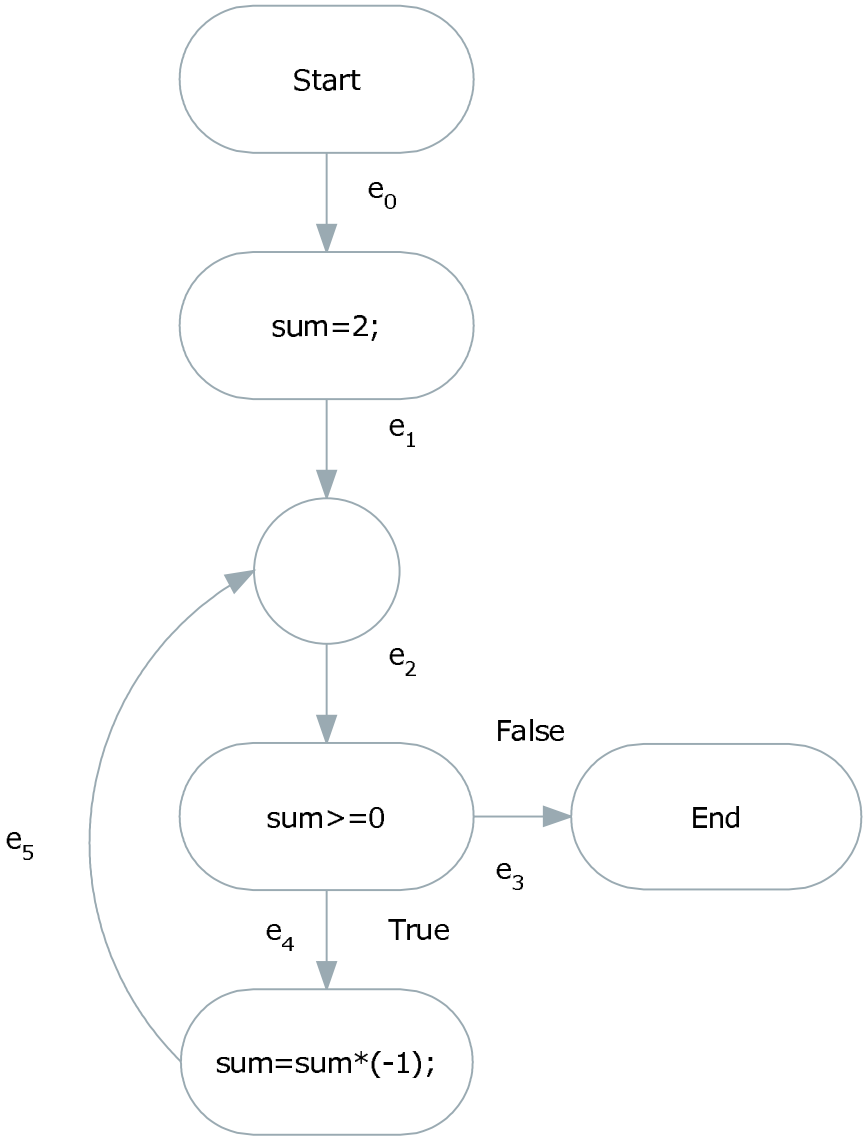


FIGURE 3.4: Control flow graph

To solve this system of equations, we begin by setting $\mathscr{T}_{\mathfrak{e}_0} = \perp$ *for all program points* $\mathfrak{e}_n$*. Then the left hand side is replaced by the right hand side of the system. The process is iterated until A fixed point is reached i.e. the left hand side equals*

$$
\begin{aligned}
&\mathscr{T}_{\mathfrak{e}_0}(\varphi_{sign}) = [sum \mapsto \bot] \\
&\mathscr{T}_{\mathfrak{e}_1}(\varphi_{sign}) = [sum \mapsto \alpha(2)] \\
&\mathscr{T}_{\mathfrak{e}_2}(\varphi_{sign}) = \mathscr{T}_{\mathfrak{e}_1}(\varphi_{sign}) \sqcup \mathscr{T}_{\mathfrak{e}_5}(\varphi_{sign}) \\
&\mathscr{T}_{\mathfrak{e}_3}(\varphi_{sign}) = \mathscr{T}_{\mathfrak{e}_2} \wedge \mathcal{H}_b[\![sum \geq 0]\!]\varphi_{sign} = TRUE \\
&\mathscr{T}_{\mathfrak{e}_4}(\varphi_{sign}) = \mathscr{T}_{\mathfrak{e}_2} \wedge \mathcal{H}_b[\![sum \geq 0]\!]\varphi_{sign} = FALSE \\
&\mathscr{T}_{\mathfrak{e}_5}(\varphi_{sign}) = \mathscr{T}_{\mathfrak{e}_4}(\varphi_{sign}[sum \leftarrow \mathcal{H}[\![sum * (-1)]\!]\varphi_{sign}])
\end{aligned}
$$

TABLE 3.13: System of equations

the right hand side.

3.4.2 Phase 2: Slicing Algorithm

This section introduces a backward slicing algorithm (Table 3.14) that uses the extracted information from *phase* 1 at each program point. While traditional slicing algorithms are typically syntactical dependency based, this property driven approach must rely on semantics dependencies and abstract dependencies. In fact, the more abstract the property, the greater the loss of precision of the syntactic approach with respect to the actual semantic. During directly relevant variable calculation we consider what is relevant as a semantic requirement. and relevant statements are collected based on the abstract dependency stated in step 2(b) of Table 3.14.

Algorithm: Property Driven Program Slicing
Input:
(1) $\mathcal{G}_P$: Statically analyzed (Phase 1) def/use graph of the program P. (2) $C = (n, V)$: slicing criterion. (3) ρ: Given property of interest.

Directly Relevant Variables ($R^0_{(C,\rho)}$)

(1) The set of directly relevant variables at slice node, n, is simply the slice set, V.

(2) The set of directly relevant variables at every other node i, is defined in terms of the set of directly relevant variables of all nodes j leading directly from i to j ($i \longrightarrow_{\mathcal{G}_{P'}} j$) in $\mathcal{G}_P$. $R^0_{(C,\rho)}(i)$ contains all variables x such that, either

$$\begin{cases} (a)\ x \in R^0_{(C,\rho)}(j) - def(i) \\ \\ (b)\ if\ (def(i) \cap R^0_{(C,\rho)}(j) \neq \emptyset)\ then \\ \quad (\forall y \neq x \in use(i)) \wedge (\forall \varphi^i_\rho, \varphi^j_\rho \in \Sigma^{\mathcal{A}}) \\ \quad\quad if(\varphi^i_\rho(y) = \varphi^j_\rho(y)) \wedge (\varphi^i_\rho(def(i)) \neq \varphi^j_\rho(def(i)))\ then \\ \quad\quad\quad R^0_{(C,\rho)}(i) = R^0_{(C,\rho)}(i) \cup \{x\} \end{cases}$$

Directly Relevant Statements ($S^0_{(C,\rho)}$)

$$if\ (def(i) \cap R^0_{(C,\rho)}(j) \neq \emptyset)\ then$$
$$S^0_{(C,\rho)} = S^0_{(C,\rho)} \cup \{i\}$$

Indirectly Relevant Variables $(R^{k+1}_{(C,\rho)}, K \geq 0)$
(a) for each predicate node b in $\mathcal{G}_P{}'$ do $if\ (b \cap S^0_{(C,\rho)}) \neq \emptyset$ $B^K_{(C,\rho)} = B^K_{(C,\rho)} \cup \{b\}$ (b) $R^{K+1}_{(C,\rho)}(i) = R^K_{(C,\rho)}(i) \cup \bigcup_{b \in B^K_{(C,\rho)}} R^0_{(b,use(b),\rho)}(i)$

Indirectly Relevant Statements $(S^{k+1}_{(C,\rho)}, K \geq 0)$
$if\ (def(i) \cap R^{k+1}_{(C,\rho)}(j) \neq \emptyset)\ then$ $S^{k+1}_{(C,\rho)} = S^{k+1}_{(C,\rho)} \cup B^K_{(C,\rho)} \cup \{i\}$

TABLE 3.14: Property driven program slicing algorithm

Example 3.6. *Let us consider the following code in Table 3.15. Notice that, statements 7 and 11 to 13 can be ignored by Rule 1(a) discussed in Table 3.9.*

Table 3.16 shows the comparison between the value based slice and property driven slice with respect to slicing criterion C=(16, w) and a property $\rho = sign$. Since the property of x at statement 2 does not depend on the property of y, statement 1 is irrelevant. The property of x stays same before and after the execution of statement 8, for that reason statement 8 is also irrelevant in this context. And statement 6 and statement 10 are deleted from the slice due to the traditional slicing rules.

3.5 Correctness of Abstract Execution

In our framework the execution of a program can be described as a succession of transitions between *concrete* values/states from the variable set V

$$v_0 \rightsquigarrow v_1 \rightsquigarrow ... \rightsquigarrow v_i \rightsquigarrow ...$$

Stmt. No.	Code	x	y	l	p	m	k	c	w
1	scanf("%d", &y);	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
2	x=2*y+1	$\perp$	$\top$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
3	l=x+1;	O	$\top$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
4	p=l;	O	$\top$	E	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
5	m=x+l;	O	$\top$	E	E	$\perp$	$\perp$	$\perp$	$\perp$
6	k= m+(x%2)-m;	O	$\top$	E	E	O	$\perp$	$\perp$	$\perp$
7	if(k!=0){	O	$\top$	E	E	O	O	$\perp$	$\perp$
8	x=p+1;	O	$\top$	E	E	O	O	$\perp$	$\perp$
9	x=x+1;	O	$\top$	E	E	O	O	$\perp$	$\perp$
10	c=x+p;}	E	$\top$	E	E	O	O	$\perp$	$\perp$
	else{	E	$\top$	E	E	O	O	E	$\perp$
11	x=x-1;								
12	p=l+1;								
13	c=x-p;}								
14	w=x+p	E	$\top$	E	E	O	O	E	$\perp$
15	printf("%d", c);	E	$\top$	E	E	O	O	E	E
16	printf("%d", w);	E	$\top$	E	E	O	O	E	E

TABLE 3.15: Program P after Phase 1

where $\rightsquigarrow$ is a transition relation. On the other hand a program analysis is an *abstract execution* of the program if the execution can be described in a *property space* $\mathcal{D}_\rho$. The *property space* $\mathcal{D}_\rho$ should be a complete lattice $\langle \mathcal{D}^\rho, \sqsubseteq \rangle$, i.e., a set $\mathcal{L}$ with an ordering relation $\sqsubseteq \subseteq \mathcal{D}_\rho \times \mathcal{D}_\rho$ such that each subset of $\mathcal{D}'_\rho \subseteq \mathcal{D}_\rho$, has a least upper bound $\sqcup\mathcal{D}'_\rho$ and greatest lower bound $\sqcap\mathcal{D}'_\rho$ in $\mathcal{D}_\rho$. Instead of values, the analysis works with properties (abstract values with respect to property ρ) $d_i \in \mathcal{D}_\rho$ where d_i *models* the value v_i based on a *correctness relation*. $\mathscr{T} : \mathcal{D}_\rho \to \mathcal{D}_\rho$ is a function, called *transfer function*, such that,

$$d_0 \mapsto d_1 = \mathscr{T}(d_0) \mapsto \ ... \ \mapsto d_i \mapsto d_{i+1} = \mathscr{T}(d_i) \ ...$$

Definition 3.8. *(Correctness Relation)* Let $\mathcal{V}$ be the set of concrete (actual) values and $\mathcal{D}_\rho$ be the set of abstract values with respect to property ρ. A relation $\mathcal{R} \in \mathcal{V} \times \mathcal{D}_\rho$ is said to be a *correctness relation* if it satisfies the following two conditions:

1. $\forall \mathfrak{v} \in \mathcal{V} \ \forall d_1, d_2 \in \mathcal{D}_\rho$: $(\mathfrak{v}\mathcal{R}d_1) \wedge (d_1 \sqsubseteq d_2) \rightarrow (\mathfrak{v}\mathcal{R}d_2)$

2. $\forall \mathfrak{v} \in \mathcal{V} \ \mathcal{D}'_\rho \subseteq \mathcal{D} \ (\forall d \in \mathcal{D}'_\rho : (\mathfrak{v}\mathcal{R}d)) \rightarrow (\mathfrak{v}(\sqcap\mathcal{D}'_\rho))$

Stmt. No.	P	$P^{sign}_{(16,w)}$	$P_{(16,w)}$
1	scanf("%d", &y);		scanf("%d", &y);
2	x=2*y+1;	x=2*y+1;	x=2*y+1;
3	l=x+1;	l=x+1;	l=x+1;
4	p=l;	p=l;	p=l;
5	m=x+l;		m=x+l;
6	k= m+(x%2)-m;		k= m+(x%2)-m;
7	if(k!=0){		if(k!=0){
8	x=p+1;		x=p+1;
9	x=x+1;	x=x+1;	x=x+1;}
10	c=x+p;}		
	else{		else{
11	x=x-1;		x=x-1;
12	p=l+1;		p=l+1;}
13	c=x-p;}		
14	w=x+p;	w=x+p;	w=x+p;
15	printf("%d", c);		
16	printf("%d", w);	printf("%d", w);	printf("%d", w);

TABLE 3.16: Property driven slice of P, $P^{sign}_{(16,w)}$, w.r.t $\rho = sign$ and C=(16,w), and value based slice of P, $P_{(16,w)}$, w.r.t $\rho = sign$ and C=(16,w)

The first condition says that if l_1 is a valid *abstract* value for v and the analysis can produce an abstract value *abstract* l_2 such that $l_1 \sqsubseteq l_2$ then l_2 is also valid (may not be the best) *abstraction* of v. Clearly first condition allows multiple *abstract* value for some v. So the second condition determines the best abstract value out of them.

Here we are combining the various *abstract* values to select the best approximation using following rules:

1. If a value $\mathfrak{v}$ is described by both d_1 and d_2, then best approximation for $\mathfrak{v}$ is $d_1 \sqcap d_2$.(*Precise*)

2. If a value $\mathfrak{v}$ is described by either d_1 or d_2, then best approximation for $\mathfrak{v}$ is $d_1 \sqcup d_2$.(*Safe*)

Suppose we have an abstract initial value d_0 and the initial concrete value $\mathfrak{v}_0$, such that $\mathfrak{v}_0 \mathcal{R} d_0$. Given a program point, we can consider all concrete execution

paths that reach it, analogously, *abstractly execute* each of them starting from $\mathfrak{v}_0$ to compute an abstract value, and join the resulting abstract values to obtain an element of the property lattice that approximates all values possible for that program point. This is called *meet-over-paths*. It is not always possible to compute the *meet-over-paths*; therefore, we approximate its result by computing a *fixed point* of a set of *dataflow equations*. To be sure that a *fixed point* exists, we require that:

1. The *transfer function* (T) is *monotone*.
2. There is no infinite *ascending chain* in $\mathcal{D}_\rho$.

To prove the correctness of the analysis, it is sufficient to prove,

1. The initial property (*abstract*) d_0 is a correct approximation of the initial value (*concrete*) v_0 and $\mathfrak{v}_0 \; \mathcal{R} \; d_0$.
2. Each transition preserves the correctness relation, i.e.,

$$\forall \mathfrak{v}_1, \mathfrak{v}_2 \in \mathcal{V} \forall d_1, d_2 \in \mathcal{D}_\rho (\mathfrak{v}_1 \rightsquigarrow \mathfrak{v}_2) \wedge (\mathfrak{v}_1 \mathcal{R} d_1) \wedge ((d_1) = d_2) \rightarrow \mathfrak{v}_2 \mathcal{R} d_2$$

   Once we prove this, an elementary induction on the length of the execution path shows that the result of *meet-over-paths* for a specific program point describes all the concrete values that can occur at that program point with respect to the correctness relation $\mathcal{R}$.

3.6 Related Work

The original definition of a program slice was presented by Weiser [129, 130, 132] in 1979 . Since then, various slightly different notions of program slices have been proposed, as well as a number of methods to compute them. In Weiser's approach, slices are computed by computing consecutive sets of transitively relevant statements, according to data flow and control flow dependencies. Only statically available information is used for computing slices; hence, this type of slice is referred to as a static slice. Since then, program slicing has grown as a

field and an amazing number of papers have been published that present different forms of program slicing, algorithms to compute them, and applications to software engineering.

An alternative method for computing static slices was suggested by Ottenstein and Ottenstein [100], who restate the problem of static slicing in terms of a reachability problem in a program dependence graph (PDG). [81][54] Horwitz et al. [76] extended the PDG based algorithm to compute inter-procedural slices on the System Dependence Graph (SDG). The authors demonstrated that their algorithm is more accurate than the original inter-procedural slicing algorithm by Weiser [132], because it accounts for procedure calling contexts. Recent improvements of algorithms to compute slices through graph reachability are presented in [106].

Yet another approach was proposed by Bergeretti and Carr [11], who define slices in terms of information-flow relations, which are derived from a program in a syntax-directed fashion. The slices mentioned so far are computed by gathering statements and control predicates by way of a backward traversal of the programs control flow graph (CFG) or PDG, starting at the slicing criterion. They were the first to define the notion of a forward static slice, although Reps and Bricker [107] were the first to use this terminology.

Venkatesh [58] presents formal definitions of several types of slices in terms of denotational semantics.Venkatesh introduced a simple procedural language L and claims to formally define the semantics of a variety of already existing forms of slice as well as introducing some of his own. He distinguishes three independent dimensions according to which slices can be categorized: static vs. dynamic, backward vs. forward, and closure vs. executable. These include the Dynamic backward clousure slice, the dynamic backword executable slice, static backward closure slice, static backword executable slice all of which, unlike Weiser's definition and functions.

Hausler [71] states the same definition of a slice as Weiser but he has written the slicing algorithm in functional language. His algorithm is a dataflow algorithm and appears, like Venkatesh, to be another formulation of Weiser's Algorithm. The strength of his work lies in the fact that he expresses a slicing algorithm without explicitly mentioning data and control dependence but they are, nevertheless encoded in his algorithm.

Reps and Yang [108] illustrate the relationship between the execution behavior of a program and the execution behavior of its slices. They state the Slicing Theorem and the Termination Theorem. The Slicing Theorem demonstrates that a slice captures a portion of a program's behavior in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each element of the slice. The Termination Theorem demonstrates that if a program is decomposed into (two or more) slices, the program halts on any state for which all the slices halt.

Binkley et al in [18] proposed a formal framework, represents different forms of slicing by means of a pair: a syntactic preorder, a function from slicing criteria to semantic equivalences. The preorder fixes a syntactic relation between the program and its slices. In standard slicing, this relation represents the fact that slices are obtained from the original program by removing zero or more statements. This preorder is called traditional syntactic ordering. The function fixes the semantic constraints that a subprogram has to respect in order to be a slice of the original program. The equivalence relation returned by the function is uniquely determined by the form of slicing and by the chosen slicing criterion.

Hong et.al [74] proposes a new approach to program slicing based on abstract interpretation and model checking. They extends static slicing with predicates and constraints by using as the program model an abstract state graph, which is obtained by applying predicate abstraction to a program, rather than a flow graph. This leads to a program slice that is more precise and smaller than its static counterpart. They develop a method for performing abstract slicing and show how abstract slicing is reduced to a least fixpoint computation over formulas in the branching time temporal logic CTL.

Recently, Isabella Mastroeni and Damiano Zanardini [91] discuss the relation between program slicing and data dependencies in Abstract Interpretation frame work. They introduce the notion of semantic dependency and abstract dependency. In this framework, since it is possible to choose dependency in the syntactic or semantic sense, thus leading to compute possibly different, smaller slices. Moreover, the notion of abstract dependency, based on properties instead of exact data values, is investigated in its theoretical meaning.

[90] extends the formal framework proposed by Binkley et al. [18] with three forms of abstract slicing, static, dynamic and conditioned. Authors show that all

existing forms are instantiations of their corresponding abstract forms and enrich the existing slicing technique hierarchy by inserting these abstract forms of slicing. Next in the literature, an algorithmic approach is provided for extracting abstract slices. The idea is to define a notion of abstract state which observes variables of interest by means of abstract properties. These states are used for analyzing the evolution of the properties of variables of interest instead of their values. In order to perform the evolution analysis, an abstract state graph (ASG) is constructed, whose vertices are abstract states and which models program executions at some level of abstraction. Then to remove all the statements not relevant for the properties of interest At this point, a technique for pruning the ASG is proposed. The algorithm is split into two modules: the simple approach, used for abstract static slicing, and the extended approach, composed of several applications of the simple one, which is used for abstract conditioned slicing.

Srihari Sukumaran et al.[122] proposed an extension of the classical program dependence graph(PDG) called the dependence conditiongraph (DCG). The DCG is obtained by adding to each PDG edge an annotation whose semantic interpretation encodes the condition under which the dependence represented by that PDG edge actually arises in a program execution. This semantic interpretation can be naturally extended to PDG paths. Based on this foundation abstract program slicing is extended in [37] by transforming the semantics-based abstract PDG into an semantics-based abstract Dependence Condition Graph (DCG) that enables to identify the conditions for dependence between program points.

Different applications of static slicing have been proposed in the literature, with some variants on the original definition. For example, Gallagher and Lyle [57] introduced the concept of decomposition slicing and discussed its application to software maintenance. A decomposition slice is defined with respect to a variable v, independently of any program point. It is given by the union of the static slices computed with respect to the variable v and all possible program points p. Weiser's slicing criterion only provides the end point and the set of output variables of the function to be extracted. Lanubile and Visaggio [84] added the set of input variables of the searched function to the slicing criterion. They introduced the notion of transform slice, as the slice that computes the values of the output variables at a given program point from the values of the input variables. The computation of a transform slice is similar to the computation of a static backward slice but stops as soon as the statements that define values for the input variables

are included in the slice. On the other hand, Cimitile et al. [26][27] defined a different slicing criterion including both the start and the end statements of the function to be extracted. The slice is computed between these two statements that form a one-in/ one-out subgraph of the CFG. The authors also defined a method to identify the slicing criterion from the specification of the searched function expressed in terms of a precondition and a postcondition.

Different program slicing surveys have also been published [19][79], as well as journal special issues [56]. A detailed survey is found in [126] by Frank Tip. His survey presents an overview of program slicing, including the various general approaches used to compute slices, as well as the specific techniques used to address a variety of language features such as procedures, unstructured control flow, composite data types and pointers, and concurrency. Horwitz and Reps [75] present a survey of the work that has been done at the University of Wisconsin-Madison on slicing, differencing, and integration of single-procedure and multi-procedure programs as operations on PDGs.

3.7 Conclusions

We further extend the previously introduced theoretical framework [129, 130, 132]. The proposed slicing algorithm has some significant advantages over the traditional slicing algorithms.

On the practical side, property driven program slicing is interesting since, in general, the slicing based on a property of some variables is smaller than the slicing technique based on the exact value of the same variables, since, properties propagate less than concrete values, some statements might affect the values but not the property. This can make debugging and program understanding tasks easier, since a smaller portion of the code has to be inspected when searching for some undesired behavior.

Since slicing is closely related to the calculus of dependencies [23], which, in turn, represents one of the basic notions in information flow [112], closely related work can be found in the abstract non-interference [60] theory where, the notion of non-interference is relaxed, in the sense that flows are only detected when they affect a property, the one which can be seen by an attacker, whose observational power is limited, rather than the concrete value of data. Due to this, a program

is more likely to satisfy abstract non-interference than standard non-interference, since some concrete flows are not really harmful at the abstract level.

Chapter 4

Watermarking Relational Databases

Watermarking is a widely used technique to embed additional but not visible information into the underlying data with the aim of supporting tamper detection, localization, ownership proof, and/or traitor tracing purposes. Watermarking techniques apply to various types of host content. Here, we concentrate on relational databases. Rights protection for such data is crucial in scenarios where data are sensitive, valuable and nevertheless they need to be outsourced.

Unlike encryption and hash description, typical watermarking techniques modify the ordinal data and inevitably cause permanent distortion to the original ones and this is an issue when integrity requirement of data are required. However some applications in which relational data are involved cannot tolerate any permanent distortions and data's integrity needs to be authenticated. To meet this requirement. In this chapter we further strengthen this approach and propose a distortion free watermarking algorithm [15, 15, 15] for relational databases and discuss it in abstract interpretation framework proposed by Patrick Cousot and Radhia Cousot [39, 42, 43].

In Section 4.1, we compare database watermarking with digital multimedia watermarking. Basic database watermarking process along with the requirements and classifications are stated in Section 4.2. In Section 4.3 we introduce the formal definitions that we are going to use in the remaining sections. The actual watermarking process is introduced in Section 4.4. We illustrate database authentication issues in Section 4.5. Section 4.6 discuss the robustness of the algorithm to handle

different watermarking attacks. In Section 4.7 states the most relevant existing database watermarking techniques. In Section 4.8, we conclude by discussing the main advantages of our scheme.

4.1 Watermarking, Multimedia vs Database

Digital multimedia watermarking technology was suggested in the last decade to embed copyright information in digital objects such images, audio and video. Most watermarking research concentrated on watermarking multimedia data objects such as still images [111] and video [70, 83, 102] and audio [5, 10, 85]. However, the increasing use of relational database systems in many real-life applications created an ever increasing need for watermarking database systems. As a result, watermarking relational database systems is now merging as a research area that deals with the legal issue of copyright protection of database systems. Techniques developed for multimedia data cannot be directly used for watermarking relational databases, because relational and multimedia data differ in a number of important respects [4]:

- A multimedia object consists of a large number of bits with considerable redundancy. Thus, the watermark has a large cover in which to hide. A database relation consists of tuples, each of which represents a separate object. The watermark needs to be spread over these separate objects.
- The relative spatial/temporal positioning of various pieces of a multimedia object typically does not change. Tuples of a relation, constitute a set, and there is no implied ordering between them.
- Multimedia objects typically remain intact; portions of an object cannot be dropped or replaced arbitrarily without causing perceptual changes in the object. On the other hand, tuple insertions, deletions and updates are the norm in the database setting.

4.2 Basic Watermarking Process

Database watermarking consists of two basic processes: watermark insertion and watermark detection, as illustrated in Figure 4.1. For watermark insertion, a key

is used to embed watermark information into an original database so as to produce the watermarked database for publication or distribution. Given appropriate key and watermark information, a watermark detection process can be applied to any suspicious database so as to determine whether or not a legitimate watermark can be detected. A suspicious database can be any watermarked database or innocent database, or a mixture of them under various database attacks.

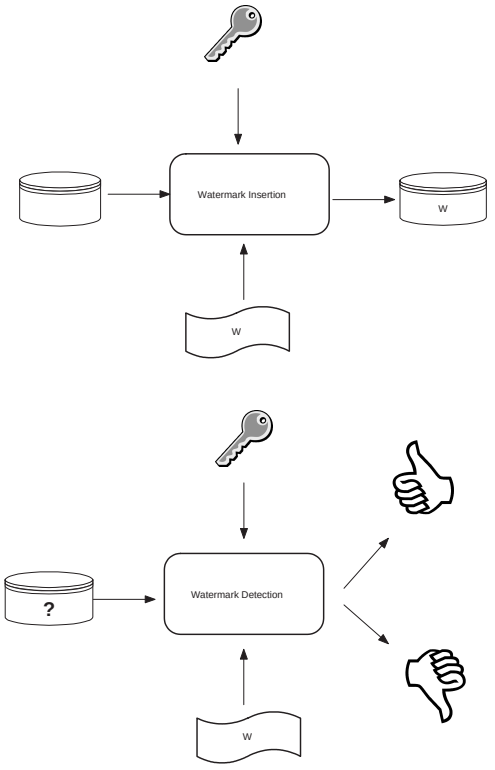


FIGURE 4.1: Basic database watermarking process

The motivation for database watermarking is to protect databases, especially those published online as parametric specifications, surveys or life sciences bio-metric data, from tampering and pirated copies. Here are some constraints of database watermarking include:

- *Watermark selection:* the insertion of the mark does not destroy the value of the Work, i.e., it is still useful for the intended purpose and it is difficult for an adversary to remove or alter the mark beyond detection without destroying this value. If the Work to be watermarked cannot be modified without losing its value then a watermark cannot be inserted. Thus, an important first step is inserting a watermark, i.e., by altering it, is to identify changes that are acceptable. Naturally, the nature and level of such change is dependent upon

the application for which the data is to be used. Clearly, the notion of value or utility of the data becomes thus central to the watermarking process. At the same time, the concept of value of watermarked Works is necessarily relative and largely influenced by each semantic context it appears in.

- *Distortion:* It is often hard to define the available bandwidth for inserting the watermark directly. Instead, allowable distortion bounds for the input data can be defined in terms of some metrics. If the watermarked data satisfies the metrics, then the alterations induced by the insertion of the watermark are considered to be acceptable. One such simple yet relevant example for numeric data, is the case of maximum allowable mean squared error (MSE), in which the usability metrics are defined in terms of mean squared error tolerances as $(s_i - v_i)^2 < t_i$, $\forall i = 1, ..., n$ and $\sum(s_i - v_i)^2 < t_{max}$, where $\mathbb{S} = \{s_1, ..., s_n\} \subset R$, is the data to be watermarked, $\mathbb{V} = \{v_1, ..., v_n\}$ is the result, $T = \{t_1, ..., t_n\} \subset \mathbb{R}$ and $t_{max} \in \mathbb{R}$ define the guaranteed error bounds at data distribution time. In other words $\mathbb{T}$ defines the allowable distortions for individual elements in terms of MSE and t_{max} its overall permissible value.

- *Uniqueness:* each value must be unique.

- *Scale:* the ratio between any two number before and after the change must remain the same.

- *Classification:* the objects must remain in the same class (defined by a range of values) before and after the watermarking.

4.2.1 Classification Model

The existing database watermarking schemes can be classified along various dimensions [86], including:

- *Data type:* Different schemes are designed for watermarking different types of data, including numerical data and categorical data.

- *Distortion to underlying data* While some watermarking schemes inevitably introduce distortions/errors to the underlying data, others are distortion-free.

- *Sensitivity to database attacks* A watermarking scheme can be either robust or fragile to database attacks. A scheme is robust (fragile, respectively) if it is difficult to make an embedded watermark undetectable (unchanged,respectively) in database attacks, provided that the attacks do not degrade the usefulness of the data significantly.

- *Watermark information* The watermark information that is embedded into a database can be a single-bit watermark, a multiple-bit watermark, a fingerprint, or multiple watermarks in different watermarking schemes.

- *Verifiability* A watermark solution is said to be private if the detection of a watermark can only be performed by someone who owns a secret key and can only be proven once to the public (e.g., to the court). After this one-time proof, the secret key is known to the public and the embedded watermark can be easily destroyed by malicious users. A watermark solution is said to be public if the detection of a watermark can be publicly proven by anyone, as many times as necessary.

- *Data structure* Different watermarking schemes are designed to accommodate different structural information of the underlying data, including relational databases (with or without primary keys), data cubes, streaming data, and XML data.

4.2.2 Requirements of Database Watermarking

Watermarking database systems have some typical requirements that differ from those required for watermarking digital image and audio systems. The watermarked database must maintain the following properties:

- *Usability* That amount of change in the database caused by the watermarking process should not result in degrading the database and making it useless. The amount of allowable change differs from one database to another, depending on the nature of stored records.

- *Robustness* Watermarks embedded in the database should be robust against attacks to erase them. That is, the database watermarking algorithm must be developed in such a way to make it difficult for an adversary to remove

or alter the watermark beyond detection without destroying usability of the database.

- *Blindness* Watermark extraction should neither require the knowledge of the original un-watermarked database nor the watermark itself. This property is critical as it allows the watermark to be detected in a copy of the database relation, irrespective of later updates to the original relation.

- *Structure* A database is made of inter-related tuples. The tuples that are joined before the watermarking process should not be altered during watermarking. Moreover, scale and classification must be considered during the watermarking process since they have impact on the semantics of the database.

- *Security* Choice of the watermarked tuples, attributes, bit positions should be secret and be only known through the knowledge of a secret-key. Owner of the database should be the only one who has knowledge of a secret-key.

4.3 Preliminaries

We shall use the capital letters at the beginning of the alphabet to denote single attributes $(A, B...)$, and $dom(A)$ will be the domain of the attribute A. For sets of attributes we shall use the letters at the end of the alphabet $(X, Y...)$, and $dom(X)$ will be the cartesian product of all the attribute domains $A \in X$. $R, S...$ will denote relational schemes (sets of attributes). $\mathcal{A}(R)$ is a set of attributes and $\mathcal{T}(R)$ is the set of tuples, over which relation schema R is defined. Relations, i.e, instances of relational schemes, will be denoted by small letters such as $r, s...$ and tuples by $t, u....$ To emphasize that the attribute A belongs to the relation r, we shall use the notation $r.A$. The value of an attribute A in a tuple t, will be represented by $t[A]$. A relational table R is a function,

$$\forall t \in \mathcal{T}(R) \wedge \forall A \in \mathcal{A}(R) : \mathcal{T}(R) \times \mathcal{A}(R) \rightarrow t[A] \in dom(A)$$

The symbol $\times$ stands for the usual cartesian product.

Example 4.1. *Consider the EMPLOYEE table (Table 4.1) as example,*

emp_no	emp_name	emp_rank
100	John	Manager
101	David	programmer
103	Albert	HR

TABLE 4.1: EMPLOYEE relation

- $\mathcal{T}(EMPLOYEE)$: $\{t_1, t_2, t_3\}$.
- $\mathcal{A}(EMPLOYEE)$: $\{emp\_no, emp\_name, emp\_rank\}$.
- *EMPLOYEE:*
  $t_1(emp\_no) = 100;\ t_1(emp\_name) = Jhon;\ t_1(emp\_rank) = Manager$
  $t_2(emp\_no) = 101;\ t_2(emp\_name) = David;\ t_2(emp\_rank) = Programmer$
  $t_3(emp\_no) = 103;\ t_3(emp\_name) = Albert;\ t_3(emp\_rank) = HR$

Now, let us consider the definition of watermarking in case of relational databases,

Definition 4.1. *(Watermarking)* A watermark W for a relation R is a predicate such that $W(R)$ is true and the probability of $W(R')$ being true with $R' \in \wp(\mathcal{T}(R') \times \mathcal{A}(R'))\backslash R$ is negligible.

4.4 Distortion Free Database Watermarking

Specifying only allowable change limits on individual values, and possibly an overall limit, fails to capture important *semantic features* associated with the data, especially if data are structured. Consider for example, the *age* data in an Indian context. While a small change to the age values may be acceptable, it may be critical that individuals that are younger than 21 remain so even after watermarking if the data will be used to determine behavior patterns for under-age drinking. Similarly, if the same data were to be used for identifying legal voters, the cut-off would be 18 years. In another scenario, if a relation contains the start and end times of a web interaction, it is important that each tuple satisfies the condition that the end time be later than the start time. For some other application it may be important that the relative ages, in terms of which one is younger, not change. It is clear from the above examples, simple bounds on the change of numerical values are often not sufficient to prevent side effects of a watermarking operation.

Our proposed watermarking technique is partition based. The partitioning can be seen as a virtual grouping, which neither change the value of the table's elements nor their physical positions. This partitioning phase is interpreted in abstract interpretation framework as *relational table abstraction*. Instead of inserting the watermark directly to the database partition, we treat it as an abstract representation of that concrete partition, such that any change in the concrete domain reflects in its abstract counterpart. This is called *partition abstracting*. The main idea is to generate a image (binary [15] or grey scale [16]) of the partition as a watermark of that partition, that serves as ownership proof (certificate) as well as tamper detection, namely *authentication* phase. The overall idea is depicted in Figure 4.2.

4.4.1 Partitioning

In this section we will describe three partitioning algorithms. Two of them have been introduced in [15][14], here we will propose a third, the most general partitioning algorithm and will describe the *abstraction* associated with each partitioning scheme.

4.4.1.1 Partition Based on Categorical Attribute

Let R be a given relational data table and $C \in \mathcal{A}(R)$ be a categorical attribute. The (finite) value set $\mathcal{V} \subseteq dom(C)$ is the set of values of C that are actually present in R. We can partition the tuples in R by grouping the values of attribute C as

$$P = \{[v_i] : 1 \leq i \leq N\}, \text{ where } \forall t \in \mathcal{T}(R) : t[C] = v_i \Leftrightarrow t \in [v_i].$$

The frequency q_i of v_i is the number of tuples in $[v_i]$. The data distribution of C in R is the set of pairs $\tau = \{(v_i, f_i) | 1 \leq i \leq N\}$. So the entire database can be partitioned into N fixed mutual exclusive areas based on each categorical value v_i.
Abstraction:
The above concept leads to an abstraction as depicted in Figure 4.3.

Given a relation R a categorical attribute $C \in \mathcal{A}(R)$ and $P = \{[v_i] : 1 \leq i \leq N\}$, for each set $S \subseteq R$, We can define a concretization map γ_x as follows:

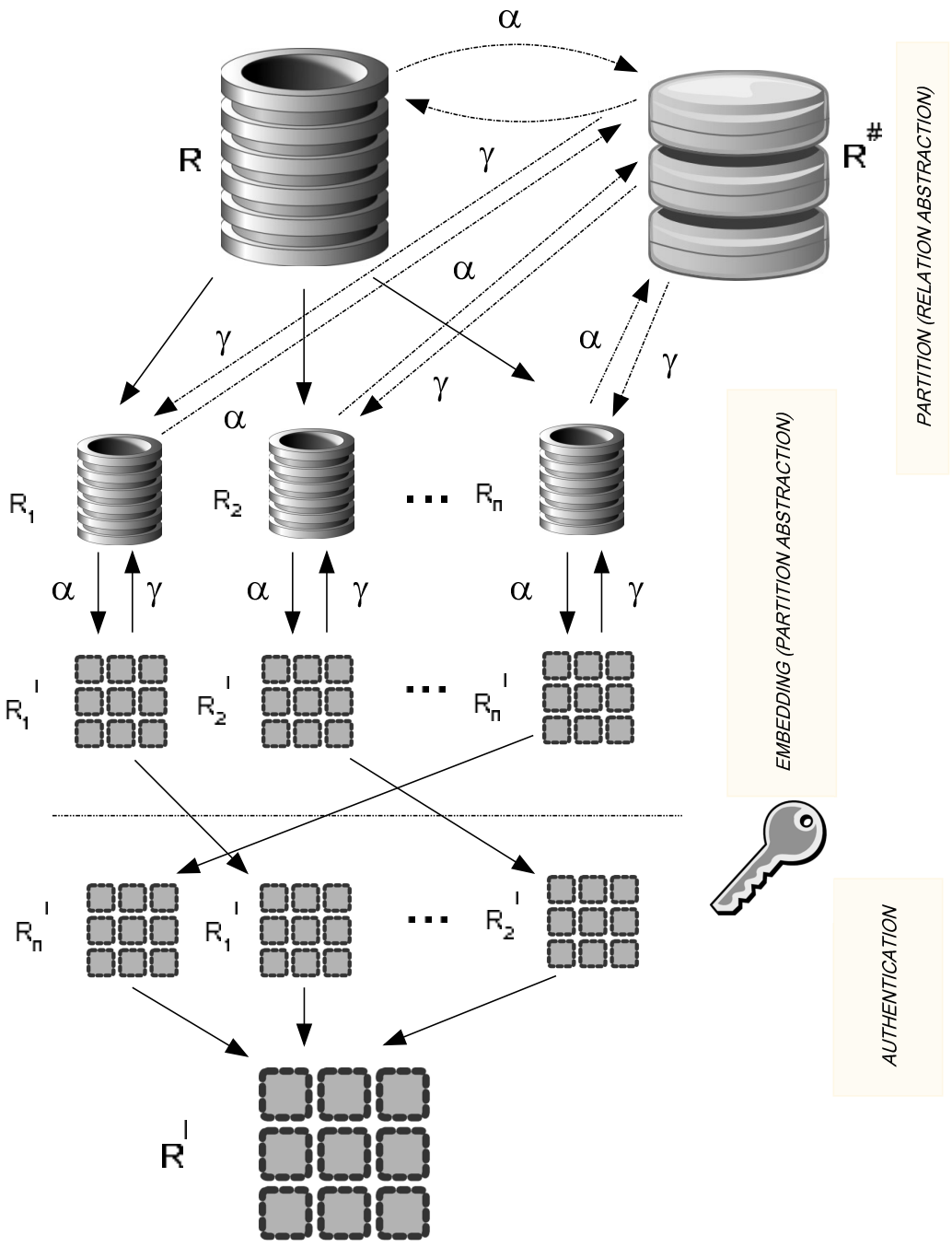


FIGURE 4.2: Block diagram of the over all process

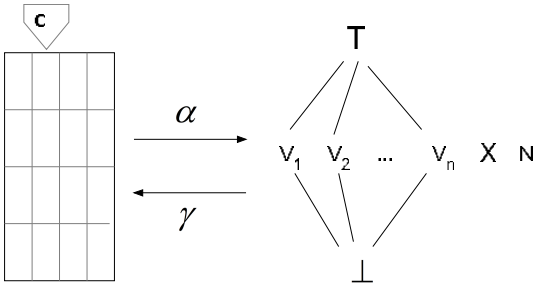


FIGURE 4.3: Table Abstraction (Galois Connection).

$$\begin{cases} \gamma_x(v_i, h) & = S \subseteq R \mid \forall t \in S : t \in [v_i] \wedge \text{size of S is h} \\ \gamma_x(\top) & = T \\ \gamma_x(\bot) & = \emptyset \end{cases}$$

The best representation of a set of tuples with attribute C is captured by the corresponding abstraction function α_x :

$$\alpha_x(S) = \begin{cases} (v_i, h) & \text{if } \forall t \in \mathcal{T}(S) : t[C] = v_i \wedge \text{size of S is h} \\ \top & \text{if } \exists t_1, t_2 \in S : t_1[C] \neq t_2.x \\ \bot & \text{if } S = \emptyset \end{cases}$$

We may prove that (α_x, γ_x) form a Galois insertion [8] with α_x monotone and γ_x weakly monotone, i.e. $(v, u) \leq (v, m) \Rightarrow (\cup\gamma(v, u)) \subseteq (\cup\gamma(v, m))$.

4.4.1.2 Secret Partitioning

In this section we present a data partitioning algorithm that partitions the relational data table based on a secret key $\Re$ with P as the primary key attribute and N is the number of tuples in R. R is partitioned into m non overlapping partitions, $[S_0], ..., [S_{m-1}]$, such that each partition $[S_i]$ contains on average $(\frac{N}{m})$ tuples from R. Partitions do not overlap, i.e., for any two partitions $[S_i]$ and $[S_j]$ such that $i \neq j$ we have $[S_i] \cap [S_j] = \emptyset$. In order to generate the partitions, for each tuple $r \in \mathcal{T}(R)$, the data partitioning algorithm computes a message authenticated code (MAC) using HMAC.[73]

Using the property that secure hash functions generate uniformly distributed message digests this partitioning technique places $(\frac{N}{m})$ tuples, on average, in each partition. Furthermore, an attacker cannot predict the tuples-to-partition assignment without the knowledge of the secret key $\Re$ and the number of partitions m which are kept secret. Keeping it secret makes it harder for the attacker to regenerate the partitions. The partitioning algorithm is described in Table 4.2.

Abstraction:

Consider the lattice $A = \langle \mathbb{N}, \bigcup\{\bot, \top\}, \sqsubseteq \rangle$, where $\bot \sqsubseteq i \sqsubseteq \top$ and $\forall$ i, j $\in \mathbb{N}$, $i \neq j$, i and j are uncomparable with $\sqsubseteq$. The lattice is shown in Figure 4.4.

$get\_partitions(R, \Re, m)$
for each tuple $r \in \mathcal{T}(R)$ do $partition \leftarrow HMAC(\Re \mid r[P])\ mod\ m$ insert r into $S_{partition}$ return$(S_0, ..., S_{m-1})$

TABLE 4.2: Secret partitiong

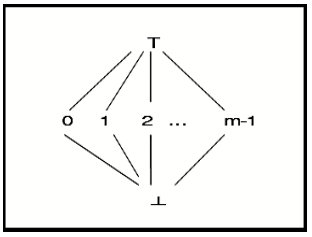


FIGURE 4.4: Lattice of the abstract domain

Given R and m partitions $\{[S_i], 0 \leq i \leq (m-1)\}$, for each set $T \subseteq R$, and given a set of natural number $i \in \mathbb{N}$, we can define a concretization map γ as follows:

$\gamma(\top) = R$
$\gamma(\bot) = \emptyset$

$$\gamma(i) = \begin{cases} T \subseteq R & \text{if } \forall t \in T : i = HMAC(\Re|t[P])\ \ mod\ \ m \\ \emptyset & \text{Otherwise} \end{cases} \tag{4.1}$$

The best representation of a set of tuples is captured by the corresponding abstraction function α :

$$\alpha(T) = \begin{cases} \bot & \text{if } S = \emptyset \\ i & \text{if } \forall t \in T : HMAC(\Re \mid t[P])\ \ mod\ \ m = i \\ \top & Otherwise \end{cases} \tag{4.2}$$

The two functions α and γ described above yield a Galois connection [40] between R and the lattice depicted in Figure 4.4. The main advantage of this partitioning is that, it is not limited to any particular type of attribute, like categorical or numerical attribute.

4.4.1.3 Partitioning Based on Pattern Tableau

Using the intersection operator over tuples we could build the tuples lattice of a relation. A closed tuple will thus subsume all tuples agreeing on the same values, i.e. the values of non empty variables in the closed tuple. This notion of set of tuples agreeing on the same values for a given set of attributes X has already been defined in database theory for horizontal decomposition purposes [101]. Let's consider the following definitions to define the partitioning.

Definition 4.2. *(Pattern tableau)* A pattern tableau $R^{\#}$ with all attributes from $\mathcal{A}(R)$, where each row $t_p \in \mathcal{T}(R^{\#})$ and each attribute $A \in \mathcal{A}(R)$, $t_p[A]$ is either:

- a constant $a \in dom(A)$.
- an empty variable $\top$ which indicates that the attribute does not contribute to the pattern.

Definition 4.3. *(X-complete property)* The relation r is said to be X-complete if and only if $\forall t_1, t_2 \in r$ we have $t_1[X] = t_2[X]$.

Informally, a relation is X-complete if all tuples agree on the attributes X.

Definition 4.4. *(X-complete-pattern)* We call X-complete-pattern of an X-complete relation r, denoted by $\mathscr{P}(X, r)$, the pattern tuple on which tuples of r agree.

since r is X-complete, its X-complete-pattern defines at least the attributes in X i.e. those attributes do not have the $\top$ value.

Definition 4.5. *(X-complete horizontal decomposition)* The set of all X-complete fragment relations of r, $\mathscr{R}_X(r)$ is defined formally as $\mathscr{R}_X(r) = \{r' \subseteq r | r \ \ is \ \ X - complete\}$.

To denote that a tuple $t \in r'$ satisfies a particular row (pattern) $t_p \in \mathcal{T}(r^{\#})$, we use the symbol $t[A] \asymp t_p[A]$, iff $\forall A \in \mathcal{A}(r)$ either $t[A] = t_p[A]$ or $t_p[A] = \top$.

Definition 4.6. *(Set of X-patterns)* The set of all X-complete-patterns of an X-complete decomposition, $\Gamma(X, r)$ is formally defined as $\Gamma(X, r) = \{\mathscr{P}(X, r') | r' \in \mathscr{R}_X(r)\}$.

T	A	B	C	D	E	F
t_1	a_1	b_1	c_1	d_1	e_1	f_1
t_2	a_1	b_1	c_1	d_1	e_2	f_1
t_3	a_2	b_1	c_2	d_2	e_2	f_1
t_4	a_2	b_1	c_2	d_2	e_3	f_1
t_5	a_2	b_2	c_2	d_2	e_1	f_2
t_6	a_2	b_2	c_2	d_1	e_1	f_2
t_7	a_2	b_2	c_1	d_1	e_1	f_2
t_8	a_2	b_2	c_1	d_2	e_1	f_2
t_9	a_1	b_2	c_2	d_1	e_2	f_2
t_{10}	a_1	b_2	c_2	d_1	e_1	f_2

TABLE 4.3: An instance relation r of the schema R

Attributes X are defined in all X-complete-patterns. Some other attributes might also be defined.

Example 4.2. *Consider Table 4.3,*

- ☐ *The $AB-$complete horizontal decomposition of r is* $\mathscr{R}_{AB}(r) = \{\{t_1, t_2\}, \{t_3, t_4\}, \{t_5, t_6, t_7, t_8\}, \{t_9, t_{10}\}\}$.
- ☐ *The set of $AB-$complete patterns is* $\Gamma(AB, r) =$
  $\{(a_1, b_1, c_1, d_1, \top, f_1);$
  $(a_2, b_1, c_2, d_2, \top, f_1);$
  $(a_2, b_2, \top, \top, e_1, f_2);$
  $(a_1, b_2, c_2, d_1, \top, f_2)\}$.

Partitions are the $X-$*complete* horizontal decomposition of R, $\mathscr{R}_X(R)$, where $X \subseteq A$, such that the following two conditions must be satisfied,

1. $\bigcup_{\forall r \in \mathscr{R}_X(R)} r = R$
2. $\bigcap_{\forall r \in \mathscr{R}_X(R)} r = \emptyset$

TABLE 4.4: Partitioning conditions

Example 4.3. *Table 4.5, Table 4.6, Table 4.7 and Table 4.8 are the partitions of Table 4.3 based on the set of $AB-$complete patterns, $\Gamma(AB, r)$.*

$\mathscr{R}_{a_1b_1}(r)$	A	B	C	D	E	F
t_1	a_1	b_1	c_1	d_1	e_1	f_1
t_2	a_1	b_1	c_1	d_1	e_2	f_1

TABLE 4.5: $\mathscr{P}(AB, \mathscr{R}_{a_1b_1}(r)) = (a_1, b_1, c_1, d_1, \top, f_1)$

$\mathscr{R}_{a_2b_1}(r)$	A	B	C	D	E	F
t_3	a_2	b_1	c_2	d_2	e_2	f_1
t_4	a_2	b_1	c_2	d_2	e_3	f_1

TABLE 4.6: $\mathscr{P}(AB, \mathscr{R}_{a_2b_1}(r)) = (a_2, b_1, c_2, d_2, \top, f_1)$

$\mathscr{R}_{a_2b_2}(r)$	A	B	C	D	E	F
t_5	a_2	b_2	c_2	d_2	e_1	f_2
t_6	a_2	b_2	c_2	d_1	e_1	f_2
t_7	a_2	b_2	c_1	d_1	e_1	f_2
t_8	a_2	b_2	c_1	d_2	e_1	f_2

TABLE 4.7: $\mathscr{P}(AB, \mathscr{R}_{a_2b_2}(r)) = (a_2, b_2, \top, \top, e_1, f_2)$

$\mathscr{R}_{a_2b_1}(r)$	A	B	C	D	E	F
t_9	a_1	b_2	c_2	d_1	e_2	f_2
t_{10}	a_1	b_2	c_2	d_1	e_1	f_2

TABLE 4.8: $\mathscr{P}(AB, \mathscr{R}_{a_2b_1}(r)) = (a_1, b_2, c_2, d_1, \top, f_2)$

Abstraction

Given a relation R and m partitions ($X-$*complete* horizontal decompositions of R) $\{[R_i], 1 \leq i \leq m\}$, then $r \subseteq R$, and given a set of $X-$*complete* patterns, we can define a concretization map γ as follows:

$$\gamma(\top) = R$$

$$\gamma(\bot) = \emptyset$$

$$\gamma(\mathscr{P}(X, r)) =$$

$$\begin{cases} r \subseteq R & \forall t \in T(r), \forall a \in A: \ \textit{either } t(a) = \mathscr{P}(X,r)(a) \textit{ or } \mathscr{P}(X,r)(a) = \top \\ \emptyset & \text{Otherwise} \end{cases}$$

The best representation of a set of tuples is captured by the corresponding abstraction function α :

$\alpha(r) =$

$$\begin{cases} \bot & \text{if } \ r = \emptyset \\ \mathscr{P}(X,r) & \textit{if } \forall t \in T(r), \forall a \in A: \ \textit{either } t(a) = \mathscr{P}(X,r)(a) \textit{ or } \mathscr{P}(X,r)(a) = \top \\ \top & \textit{Otherwise} \end{cases}$$

Example 4.4. *Table 4.9 shows the concrete relation instance r and associated abstract relation instance $r^{\#}$. Notice that, each tuple $t_p \in \mathcal{T}(r^{\#})$ is associated with a non-overlapping partition in r.*

T	A	B	C	D	E	F	T	A	B	C	D	E	F
t_1	a_1	b_1	c_1	d_1	e_1	f_1	$\top$	a_1	b_1	c_1	d_1	$\top$	f_1
t_2	a_1	b_1	c_1	d_1	e_2	f_1							
t_3	a_2	b_1	c_2	d_2	e_2	f_1	$\top$	a_2	b_1	c_2	d_2	$\top$	f_1
t_4	a_2	b_1	c_2	d_2	e_3	f_1							
t_5	a_2	b_2	c_2	d_2	e_1	f_2	$\top$	a_2	b_2	$\top$	$\top$	e_1	f_2
t_6	a_2	b_2	c_2	d_1	e_1	f_2							
t_7	a_2	b_2	c_1	d_1	e_1	f_2							
t_8	a_2	b_2	c_1	d_2	e_1	f_2							
t_9	a_1	b_2	c_2	d_1	e_2	f_2	$\top$	a_1	b_2	c_2	d_1	$\top$	f_2
t_{10}	a_1	b_2	c_2	d_1	e_1	f_2							

TABLE 4.9: Concrete Relation r and the corresponding abstract relation $r^{\#}$

4.4.2 Watermark Generation

We are interested in a watermark generation process starting from a partition $[R_k]$ $1 \leq k \leq n]$, in a relational database table . The partitioning can be seen as a virtual grouping which does not change the physical position of the tuples as described in the last section. Let the owner of the relation R possess a watermark key $\Re$, which will be used in both watermark generation and detection. In addition, the key should be long enough to thwart brute force guessing attacks to the key. A cryptographic pseudo random sequence generator [113] $\mathcal{G}$ is seeded with the concatenation of watermark key $\Re$ and the primary key $r[P]$ for each tuple $r \in \mathcal{T}(R_k)$, generating a sequence of numbers, through which we select a field (attribute) in $\mathcal{A}(R)$. A fixed number of MSBs (most significant bits) and LSBs (least significant bits) of the selected field are used for generating the watermark of that corresponding field. The reason behind it is: a small alteration in that field in R will affects the LSBs first and a major alteration will affects the MSBs, so the LSB and MSB association is able to track the changes in the actual attribute values. So here we make the watermark value as the concatenation of m number of MSBs and n number of LSBs such that $m + n = 8$. Our aim is to make a *grey scale* image as the watermark of that associated partition, so the value of each cell must belongs to [0..255] range. Formally, the watermark (grey scale image) R_k^I corresponding to the k^{th} partition $[R_k]$ is generated in Table 5.1,

Example 4.5. *Let us illustrate the above algorithm for a single tuple in any hypothetical partition of a table Employee* $= (emp\_id, emp\_name, salary, location, position)$*, where emp_id is the primary key which is concatenated along with the private key* $\Re$ *as in line 4 in the above algorithm to select random attributes. Here (10111111,*

$genW(R_k, \Re)$

for *each tuple* $t \in \mathcal{T}(R_k)$ do
construct a row $p \in \mathcal{T}(R_k^I)$
for $(i = 0; i < \lvert\mathcal{A}(R_k)\rvert; i = i + 1)$ do
$j = \mathcal{G}_i(\Re, r[P])\ mod\ \lvert\mathcal{A}(R_k)\rvert$
$p_i.R_k^I = (t[j]_{m\ MSBs} \mid t[j]_{n\ LSBS})_{10}\ mod\ 256$
delete the j^{th} attribute from t
endfor
endfor
return(R_k^I)

TABLE 4.10: Watermark generation

10110101, 10010101,11110111) is the generated watermark for the tuple (Bob, 10000, London, Manager), where we consider 4 MSBs concatenated with 4 LSBs. And the attribute watermark pair looks like $\{\langle Bob, 10010101\rangle, \langle 10000, 10111111\rangle, \langle London, 11110111\rangle, \langle Manager, 10110101\rangle\}$. *The entire concept is illustrated by Figure 4.5.*

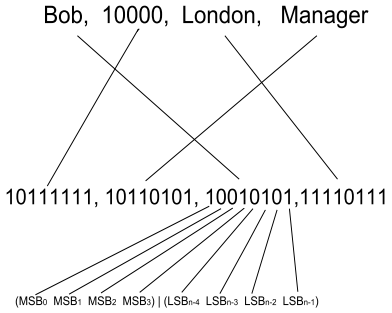


FIGURE 4.5: Watermark generation for a single tuple

The whole process does not introduce any distortion to the original data. The use of MSB LSB combination is for thwarting potential attacks that modify the data as it simply produces an integrity certificate.

4.4.2.1 Abstraction

Let us define the abstract framework behind this generation algorithm, let

- $\mathbb{R} = \{R : T \times A \to \mathbb{Z}\}$

- ☐ $\mathbb{R}^I = \{R^I : T \times A \rightarrow [0..255]\}$
- ☐ The abstraction function $\alpha : \mathbb{R} \rightarrow \mathbb{R}^I$ is defined as $\alpha(R)(t, a) = \overline{\alpha}(R(t, a))$ where $\overline{\alpha} : \mathbb{Z} \rightarrow [0..255]$.

4.4.3 Watermark Detection

A very important problem in a watermarking scheme is synchronization, that is, we must ensure, that the watermark extracted is in the same order as that generated. If synchronization is lost, even if no modifications have been made, the embedded watermark cannot be correctly verified. In watermark detection, the watermark key $\Re$ and watermark R_k^I are needed to check a suspicious partition R'_k of the suspicious database relation R'. It is assumed that the primary key attribute has not been changed or else can be recovered. Table 4.11 states the watermark detection algorithm.

$genW(R_k, \Re)$

for *each tuple* $t \in \mathcal{T}(R_k)$ do
construct a row $p \in \mathcal{T}(R_k^I)$
for $(i = 0; i < \lvert\mathcal{A}(R_k)\rvert; i = i + 1)$ do
$j = \mathcal{G}_i(\Re, r[P])$ *mod* $\lvert\mathcal{A}(R_k)\rvert$
if $p_i.R_k^I = (t[j]_{m\ MSBs} \mid t[j]_{n\ LSBS})_{10}$ *mod* 256 then
matchC = matchC + 1
endif
delete the j^{th} attribute from t
endfor
endfor
if $matchC = \omega$ then
// ω = number of rows × number of columns in R_k^I return true
else
return false
endif

TABLE 4.11: Watermark detection

The variable *matchC* counts the total number of correct matches. The authentication is checked by comparing the generated watermark bitwise. And after each match *matchC* is increased by 1. Finally, the total match is compared to the

number of bits in the watermark image R_k^I associated with partition R_k to check the final authentication.

4.5 Zero Distortion Authentication Watermarking (ZAW)

So far, we have a set of grey scale images corresponding to a data table R. Each gray scale image R_k^I (k=1 to n) is associated with n partitions R_k (k=1 to n) of R. And image R_k^I is said to be the abstraction of partition R_k. Now the authentication of database owner is necessary. We employ the zero-distortion authentication watermarking (ZAW) [133] to authenticate the table which introduces no artifact at all. Figure 4.6 describes the framework of the ZAW scheme which does not modify the host content but transforms the host into its equivalence.

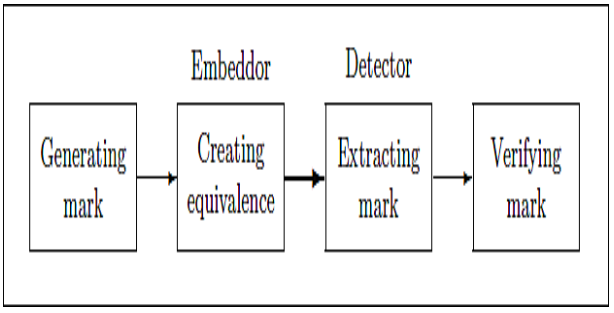


FIGURE 4.6: ZAW framework

Without loss of generality we assume that the table R is fragmented into n independent grey scale images R_1^I; R_2^I; ...;R_n^I. Each image does not depend on any other images. If we consider R as the concrete table then R^I (composition of all image fragments R_1^I; R_2^I; ...;R_n^I) can be considered as its abstract counterpart. An equivalent image can be derived from π_k using Myrvold and Ruskey's linear permutation ranking algorithm [96] by permuting the partitions in R^I. The algorithm *unrank* makes a permutation of the segments based on a secret number (M) only known to the database owner and this number can be considered as a private key of the owner. The owner can distribute the number of partitions n as public key. *unrank* in Table 4.12 can be treated as a encryption algorithm based on the private key M.

The algorithm *rank* in Table 4.13 can be treated as decryption algorithm based on the public key n.

$Unrank(n, M, \pi)$

for $(i = 0; i < n; i++)$ do
$\pi_i = i$
endfor
if$(m > 0)$ then
$swap(\pi[n-1], \pi[M \ mod \ n])$
$Unrank(n-1, \lfloor M/n \rfloor, \pi)$
endif

TABLE 4.12: Encryption

$rank(n, \pi, \pi^{-1})$

if $(n = 1)$ then
return 0
endif
$s = \pi[n-1]$
$swap(\pi[n-1], \pi[\pi^{-1}[n-1]])$
$swap(\pi^{-1}[s], \pi^{-1}[n-1])$
return (s+m * rank$(n-1, \pi, \pi^{-1})$)

TABLE 4.13: Decryption

4.6 Robustness

We analyze the robustness of our scheme by Bernoulli trials and binomial probability. Repeated independent trials in which there can be only two outcomes are called Bernoulli trials in honor of James Bernoulli (1654-1705).The probability that the outcome of an experiment that consists of n Bernoulli trials has k successes and $n - k$ failures is given by the binomial distribution

$$b(n, k, p) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad 0 \leq k \leq n$$

where the probability of success on an individual trial is given by p.

The probability of having at least k successes in n trials, the cumulative binomial probability, can be written as

$$B(n, k, p) = \sum_{i}^{k} b(n, i, p)$$

We will discuss our robustness condition based on two parameters *false hit* and *false miss*.

4.6.1 False Hit

False hit is the probability of a valid watermark being detected from non-watermarked data. The lower the false hit, the better the robustness.

When the watermark detection is applied to non-watermarked data, each $\langle MSB_m | LSB_n \rangle$ association (grey scale entry) has the probability $\frac{1}{2^8}$ to match to the corresponding entry in R^I. Assume that for a non-watermarked data partition R'_q and watermarking data partition R_k, $|\mathcal{A}(R'_q)| = |\mathcal{A}(R_k)|$, $|\mathcal{T}(R'_q)| = |\mathcal{T}(R_k)|$ and $P.R'_q = P.R_k$, i.e. both have same number of tuples, attributes and same primary keys, respectively. Let $\omega = |\mathcal{A}(R_k)| * (m + n) * |\mathcal{T}(R_k)|$ is the size of the watermark. The false hit is the probability that at least $\frac{1}{\mathfrak{T}}$ portion of ω can be detected from the non-watermarked data by sheer chance. When $\mathfrak{T}$ is the watermark *detection parameter*. It is used as a tradeoff between false hit and false miss. Increasing $\mathfrak{T}$ will make the robustness better in terms of false hit. Therefore, the false hit F_h can be written as

$$F_h = B(\omega, \lfloor \frac{\omega}{\mathfrak{T}} \rfloor, \frac{1}{2^8})$$

4.6.2 False Miss

False miss is the probability of not detecting a valid watermark from watermarked data that has been modified in typical attacks. The less the false miss, the better the robustness.

4.6.2.1 Subset Deletion Attack

For tuple deletion and attribute deletion, the $\langle MSB_m|LSB_n\rangle$ association in the deleted tuples or attributes will not be detected in watermark detection; however, the other tuples or attributes will not be affected. Therefore, all detected bit strings will match their counterparts in the watermark, and the false miss is zero.

4.6.2.2 Subset Addition Attack

Suppose an attacker inserts ς new tuples to replace ς watermarked tuples with their primary key values unchanged. For watermark detection to return a false answer, at least $\frac{1}{\mathfrak{T}}$ bit strings of those newly added tuples (which consists of $v\varsigma$ $\langle MSB_m|LSB_n\rangle$) must not match their counterparts in the watermark (which consists of ω bits). also in this case $\mathfrak{T}$ is the watermark detection parameter, used as a tradeoff between false hit and false miss. Increasing $\mathfrak{T}$ will make the robustness worse in terms of false miss. Therefore, the false miss F_m for inserting ς tuples can be written as

$$F_m = B(v\varsigma, \lfloor\frac{|\mathcal{A}(R_k)| * \varsigma}{\mathfrak{T}}\rfloor, \frac{1}{2^8})$$

The formulae F_h and F_m together, give us a measure of the robustness of the watermark.

4.7 Related Work

Watermarking in relational frameworks is a relatively young technology that has begun its maturity cycle towards full deployment in industry-level applications. A brief overview of recent research on watermark is given below based on the classification presented in the last section.

The first well-known database watermarking scheme was proposed by Agrawal and Kiernan [4] for watermarking numerical values in relational databases. The fundamental assumption is that the watermarked database can tolerate a small amount of errors: it is acceptable to change a small number of least significant bits in some numeric values; however, the value of data is significantly reduced if

a large number of the bits are changed. The basic idea is to ensure that those bit positions contain specific values determined by a secret key K. The bit pattern constitutes a watermark.

Agrawal and Kiernans scheme has been extended by Ng and Lau to watermark XML data [98]. In this scheme, the owner of the XML data is responsible for selecting the XML elements that are suitable to be locators, where a locator is defined to have a unique value that can serve as a primary key in the watermarking process, as in Agrawal and Kiernans scheme. The difference between this scheme and Agrawal and Kiernans scheme is that if a textual value of an element is selected to embed a mark bit, one of its words is chosen and replaced by a synonym function based on a well-known synonym database WordNet.

This scheme is further extended and deployed on a XML compression system. In [65] Gross-Amblard introduce interesting theoretical results investigating alterations to relational data (or associated XML) in a consumer-driven framework in which a set of parametric queries are to be preserved up to an acceptable level of distortion.The author shows if the family of sets defined by the queries is not learnable, no query-preserving data alteration scheme can be designed. In a second result, the author shows that if the query sets defined by first-order logic and monadic second order logic on restricted classes of structures with a bounded degree for the Gaifman graph or the tree-width of the structure, a query-preserving data alteration scheme exists. [12] Bertino et. al. explore issues at the intersection of two important dimensions in data-centric assurance, namely rights assessment and privacy, in the broader context of medical data. A unified framework is introduced that combines binning and watermarking techniques for the purpose of achieving both data privacy and the ability to assert rights. The framework then deploys a version of the encoding for categorical types [114] [120] by Sion et. al. in a hierarchical fashion, for the purpose of defeating a data generalization attack of concern in this framework.

In [87], Li, Guo, and Jajodia introduced a distortion-free scheme for watermarking categorical data. The purpose of fragile watermarking is not to protect copyright, but to detect and localize possible attacks that modify a distributed or published database. Guo et al. [66] proposed another fragile watermarking scheme that can further improve the precision in tamper localization, assuming that the database relation to be watermarked has numerical attributes and that the errors introduced in two least significant bits of each value can be tolerated.

Patents have been filed for several of them, including Agrawal et.al. [86] [80] and Sion et.al. [114, 115, 120] [116] [117–119].

4.8 Conclusions

As a conclusion, let us stress the main features of the watermark technique presented in this dissertation

- ☐ It does not depend on any particular type of attributes (categorical, numerical);
- ☐ It ensures both authentication and integrity.
- ☐ As it is partition based, we are able to detect and locate modifications as we can trace the group which is possibly affected when a tuple t_m is tampered;
- ☐ Neither watermark generation nor detection depends on any correlation or costly sorting among data items. Each tuple in a table is independently processed; therefore, the scheme is particularly efficient for tuple oriented database operations;
- ☐ It does not modify any database item; therefore it is distortion free.
- ☐ This watermarking process has an advantage over hash function, as this watermarking procedure does not depend on the ordering of the tuples, so it is free from false alarms, that depends on just by altering the order of the tuples.

Chapter 5

Zero-Knowledge Source Code Watermarking

With the increasing amount of program source code (most of the time in the form of bytecode) which is distributed in the web, software ownership protection and detection is becoming an issue. In particular, with multiple distributions of code, in order to prevent the risk of running fake programs, it is important to provide authentication proofs that do not overload the packages and that are easy to check. This is the aim of the so called Software Watermarking Techniques. Software watermarking embeds hidden information about ownership and integrity of the program into the code itself which can be retrieved and checked automatically on demand. In general, it is not possible to devise watermarks that are immune to all conceivable attacks; it is generally agreed that a sufficiently determined attacker will eventually be able to defeat any watermark. In our vision, watermarking is a method that does not aim to stop piracy copying, but to check the ownership of the software. Therefore in our approach watermarking is seen as an alternative to encryption as a way to support software authentication rather a tool for copyright protection.

In this chapter we focus our attention on a semantic-preserving program transformation based public key source code watermarking (asymmetric watermarking) scheme [17]which is similar in sprit to zero-knowledge proofs introduced by Goldwasser, Micali, Rackoff [63]. The main idea is to prove the presence of a watermark without revealing the exact nature of the mark.

In Section 5.1, we state different schemes of software watermarking available. In section 5.2, we introduce the formal definitions that we are going to use in the remaining sections. The actual source code watermarking algorithm is introduced in Section 5.3. Computational complexity of the technique is discussed in Section 5.4. Section 5.5 discuss the ability of the algorithm to handle different watermarking attacks. In Section 5.6 we state how we can use this algorithm to fingerprint a source code. In Section 5.7, we categorize the most relevant existing software watermarking techniques. In section 5.8, we conclude by discussing the main advantages of our scheme.

5.1 Different Software Watermarking Schemes

The methodology for software watermarking can be divided into two major types, static [47] and dynamic [34]. Systems that encode the watermark data directly in the program executable are static systems. The watermark may be stored in any part of the executable, so long as the semantics of the program are preserved. To detect the watermark, the program executable is statically analyzed by a decode function, searching for the watermark data. Instead of encoding the watermark data directly in the text of the program executable, some systems add code to the program which constructs the watermark in the runtime state of the program. Such systems were first proposed by Collberg and Thomborson [47] and are called dynamic systems. To detect the watermark, instead of analyzing the program directly, some other artifact of the program is searched such as a profile of the run time state of the program. Depending on the nature of the extraction algorithm, two types of watermarking schemes can be identified. The extraction process of *private watermarking* systems take the watermarked media, the original media, the watermark and the secret key and outputs TRUE if the watermark is actually present. In the case of *blind watermarking* systems, the extractor extracts the watermark given only the watermarked media and the key. Blind watermarking is preferable to non-blind marking, because publication of the original, unmarked media allows subsequent piracy. Traditional watermarking systems (*symmetric watermarking)* require the complete disclosure of the private watermark key in the watermark verification process. This is a major security risk, since this information is in most cases sufficient to remove the watermark and to defeat the goal of copyright protection. This problem of information leakage strongly limits the

usability of *symmetric watermarks*. Such problems could be avoided by *public-key watermarking*. Each user would have a private key to embed a watermark which everybody could verify using the corresponding public key. [49][50] presented an extension to watermarking system, in which a mark is inserted by a private key but the presence of the watermark can be checked using different (public) key watermark extraction should also be possible in case small modifications have been applied to the marked media. Such modifications can be the result of intentional attacks in order to remove the mark or the result of coding schemes (e.g. lossy compression) and errors during the transmission [50]. Schemes which are able to retrieve a watermark from a distorted media are called *robust*.

5.2 Preliminaries

In this section, we present the definitions of the primitives which are required for a formal definition of zero-knowledge watermark detection [35] [1] and that will be used in section 4 in order to properly explain our watermarking technique.

Definition 5.1. *(Semantics-preserving program transformation)* Let dom(P) be the set of input sequences accepted by a program P, out(P, I) be the output of P on input I. Let $\mathbb{T}$ be the set of transformations from programs to programs. An input output semantics-preserving transformation $T \in \mathbb{T}$ satisfies the following properties

- $\square$ dom(P) = dom(T(P)),
- $\square$ $\forall$ I $\in$ dom(P): out(P, I) = out(T(P), I).

Definition 5.2. *(Prover and Verifier)* To describe zero-knowledge proofs, we need the notion of a prover and a verifier. The prover is an agent or entity who claims the knowledge of the proof of a statement and tries to prove it. The verifier is an agent or entity who tries to learn the proof from the verifier. At the end of the interaction, called a protocol, a prover convinces the verifier about his knowledge of the proof (but not any additional knowledge). If the prover does not know the proof, he is called a cheating prover. The protocol is designed so that the verifier would not accept the proof of a cheating prover. A cheating verifier, is the one who tries to gain knowledge from the prover through the protocol executions.

Definition 5.3. *(Commitment Schemes)* A commitment scheme $(com, open)$ for the message space M and commitment space C consists of a two-party protocol com to commit to a value $m \in M$ and a protocol $open$ that opens a commitment. A commitment to a value m is denoted by $com(m, par_{com})$ where par_{com} stands for all public parameters needed to compute the commitment value. To open a commitment com the committer runs the protocol $open(com, par_{com}, sk_{com})$ where sk_{com} is the secret opening information of the committer.

The security requirements are,

- ☐ *binding (committing)* requires that a dishonest committer cannot open a commitment to another message $m' \neq m$ than the one to which he committed.
- ☐ *hiding (secrecy)* requires that the commitment does not reveal any information about the message m to the verifier.
- ☐ *homomorphic property:* Let $com(m_1)$ and $com(m_2)$ be commitments to arbitrary messages $m_1, m_2 \in M$. Then the committer can open $com(m1) * com(m2)$ to $m1 + m2$ without revealing additional information about the contents of $com(m_1)$ and $com(m_2)$.

Definition 5.4. *(Interactive Proof systems)* An interactive proof system for a set S is a two-party game between a verifier executing a probabilistic polynomial-time strategy and a prover which executes a computationally unbounded strategy satisfying:

- ☐ *Completeness:* For every $x \in S$, the verifier always accepts after interacting with the prover on common input x.
- ☐ *Soundness:* For some polynomial p, it holds that for every $x \notin S$ and every potential strategy P^*, the verifier rejects with probability at least $\frac{1}{p(|x|)}$ after interacting with P^* on common input x.

Informally, a proof is complete if an honest verifier will always be convinced of a true statement by an honest prover. A proof is sound if a cheating prover can convince an honest verifier that some false statement is actually true with only a small probability. A proof is further considered to be zero-knowledge if it satisfies the following definition.

Definition 5.5. *(Zero-knowledge)* A strategy A is zero-knowledge on (inputs from) the set S if, for every feasible strategy B^* there exists a feasible computation C^* so that the following two probability ensembles are computationally indistinguishable:

- ☐ the output of B^* after interacting with A on common input $x \in S$
- ☐ the output of C^* on input $x \in S$

the first ensemble represents the output of an actual execution of the proof system protocol, while the second ensemble (called the *simulation*) is the output of a stand-alone procedure which is not a part of any interactive system. A proof is called zero-knowledge if the output of any strategy B^* used by a cheating verifier could also be produced by the non-interactive computation C^*. In other words, whatever information can be learned by interacting with A on some input x can also be extracted from x without interacting with A.

Definition 5.6. *(Software watermarking schemes)*Let $\mathbb{P}$ is the set of programs to be watermarked. Software watermarking schemes can be defined by a tuples, $S = (G_{key}(), G_W(), E(), D())$, where

- ☐ $G_{key}()$ is a polynomial-time algorithm. On input of the security parameters, it generates the keys (K_{emb}, K_{det}), required for watermark embedding and detection.
- ☐ $G_W()$ is a polynomial-time algorithm. On input of the security parameters, it generates the watermark W.
- ☐ On input of a program $P \in \mathbb{P}$, a watermark W to be embedded and the embedding key K_{emb} the polynomial time embedding algorithm $E(P, W, K_{emb})$ outputs the watermarked program P_W.
- ☐ On input of a possibly modified watermarked program P'_W, the watermark W, the original program P and the detection key K_{det}, the detection algorithm $D(P'_W, P, W, K_{det})$ outputs a boolean value, 1 for the presence of W in P'_W relative to the reference program P and 0, otherwise.

A *symmetric* watermarking scheme needs the same key k_{wm} ($k_{emb} = k_{det} = K_{wm}$) for detection as for embedding. Watermarking schemes whose $D()$ algorithm does

not require the input of reference data W are called *blind*, in contrast to non-blind schemes.

Definition 5.7. *(Zero-knowledge watermark detection)* Let $(com, open)$ be a secure commitment scheme. A zero-knowledge watermark detection protocol ZK_DETECT for the *watermarkingscheme* $(G_{key}(), G_W(), E(), D())$ is a zero-knowledge proof of knowledge protocol between a prover P and a verifier V: The common protocol input of P and V is the stego-data P'_W, $com(W), com(P), com(K_{det})$, i.e., commitments on the watermark, the reference data and the detection key respectively, as well as the public parameters $par_{com} = (par^W_{com}, par^P_{com}, par^{K_{det}}_{com})$) of these commitments. The private input of the prover is the secret opening information of these commitments $sk_{com} = (sk^W_{com}, sk^P_{com}, sk^{k_{det}}_{com})$.

P proves knowledge of a tuple $(W, P, K, sk^W_{com}, sk^P_{com}, sk^{k_{det}}_{com})$ such that:

$$
\begin{gathered}
[(open(com(W), par^W_{com}, sk^M_{com}) = M) \bigwedge \\
(open(com(P); par^P_{com}, sk^P_{com}) = P) \bigwedge \\
(open(com(K_{key}), par^{K_{key}}_{com}, sk^{K_{key}}_{com}) = K_{key} \bigwedge \\
D(P'_W, W, P, K_{det})] = true
\end{gathered}
$$

The protocol outputs a boolean value to the verifier, stating whether to accept the proof or not.

Zero-knowledge watermark detection enables a prover to prove to an un-trusted verifier that a certain watermark is present in stego-data without revealing any information about the watermark, the reference data and the detection key.

5.3 Source Code Watermarking

The central idea of our watermarking scheme can be defined by the following steps,

- ☐ The watermark embedding process is based on a semantics-preserving program transformation using a permutation on the set of n syntactic elements of the program P.

- ☐ The watermark detection and verification process is based on a permutation that produces a scrambled version of the watermarked program by reordering m syntactic elements present in the watermarked program. This permutation is introduced to prove the ownership in a interactive zero-knowledge proof system framework.

- ☐ This watermarking scheme can be combined with a *time-stamp* mechanism that makes it robust against invertibility or ambiguity attacks.

Let us illustrate our source code watermarking scheme on C source codes, exploiting explicitly the programming language features. Let $P \in \mathbb{P}$ be the program to be watermarked. Let $Id(P) = InId(P) \cup LocId(P) \cup OutId(P)$ be the identifiers of program P, where *InId(P)* are the identifiers defined in other functions imported by P (e.g. *extern* variables),*OutId(P)* are the identifiers that are exported (global variables) to other programs interacting with P, and *LocId(P)* are the remaining local identifiers. The central idea of our watermarking scheme can be defined by the following two steps

- ☐ The watermark embedding process is based on a semantics-preserving program transformation using a permutation π on *LocId(P)*. Suppose $|LocId(P)| = n$, then *LocId(P)* is considered as n syntactic elements for watermarking P.

- ☐ The watermark detection and verification process is based on a permutation Γ, produces a scrambled version of the watermarked program P_π by reordering the functions defined in (P). Suppose there are m functions defined in (P), then we consider each function as a syntactic element, reordering them introduces compile time errors. This permutation is introduced to prove the ownership in a interactive zero knowledge proof system framework.

5.3.1 Watermark Generation

Let the owner of the program P possess a secured key $\Re$; the key should be long enough to thwart brute force guessing attacks. A cryptographic pseudo random sequence generator G is seeded with key $\Re$ and the concatenated identifiers, generating a sequence of numbers. A permutation π is chosen from the permutation group S_n on n elements (assuming $|LocId(P)| = n$) on the basis of the output generated by G [**?**]. The permutation π is considered as a security parameter for

the watermark embedding algorithm and $\pi(LocId(P))$ will be the generated watermark. Here we adopt the function unrank following the Myrvold and Ruskey's linear permutation unranking algorithm [96] to generate the permutation π as stated in Table 5.1.

genW(LocId(P), ℜ)
r= $G(\Re, Id_1 \| Id_2 \| ... \| Id_n)$ mod (n!) $LocId(P) = \{Id_1, Id_2, ..., Id_n\}$ for i=1 to n do $\pi_i = Id_i$ $Unrank(\pi, n, r)$ **return**$(\pi(LocId(P)))$

Unrank$(\pi$, n, $r)$
if $n > 0$ then
  swap (π[n-1], $\pi[r$ mod $n]$)
  $Unrank(n-1, \lfloor r/n \rfloor, \pi)$
endif

TABLE 5.1: Watermark generation algorithm

5.3.2 Watermark Embedding

The watermark embedding function is a semantics-preserving program transformer which just substitutes each identifier $LocId(P)$ by the corresponding one in $\pi(LocId(P))$. For the sake of simplicity we denote $LocId(P)$ as Id and $\pi(LocId(P))$ as Id_π. A single variable substitution can be expressed as,

$$\forall x \in Id, \exists y \in Id_\pi, [y/x]Q = ([x]Q)y.$$

The left side reads *the substitution of y for all occurrences of variable x in expression Q*. The formula states that this is equal to a new expression derived from the original one, applied as a function to argument y. The watermark embedding algorithm can be expressed as,

$$E_{\Re}\colon P_{Id} \rightarrow P_{Id_\pi}.$$

Where $E_{\Re}$ is the watermark embedding algorithm, $\Re$ the secure watermark embedding key, P_{Id} is the program to be watermarked and P_{Id_π} is the watermarked program. So far we used traditional symmetric key watermarking algorithm to generate the watermarked program P_{Id_π}. It could be a very tedious job for the attacker to select the actual permutation from S_n in worst case scenario, since the number of identifiers(n) are supposed to be huge in a software which increases the number of permutations (n!) radically.

5.3.3 Watermark Detection and Verification

Assume that software owner (*Prover*) Alice inserted her watermark W into the program by using watermarking algorithm described in section 3.2, yielding the watermarked program P_{Id_π}. The software user (*Verifier*) Bob who is expecting the proof of authentication from Alice. To do so, both Alice and Bob participate to a challenge-response protocols in a interactive zero knowledge proof system framework as follows, Let $F(P) = \{f_1, f_2, ..., f_m\}$ be the set of functions defined in program P. Alice now produces a scrambled version of her software by another permutation Γ. The central idea of this scheme (scrambling) is to alter the sequence in which the functions in $F(P)$ appear in P, by choosing a hidden permutation Γ from the permutation group S_m. We denote the scrambled program obtained in this way by P_{F_Γ}. Alice sets up a public directory where she publishes P_{Id_π} along with P_{F_Γ}. We assume that the software consists of a large number of functions so that the size of the permutation group S_m, is large enough to make the permutation difficult to guess by brute force. Notice that P_{F_Γ} is useless with respect to program execution, because altering the sequence of order of functions in P_{Id_π} might yield to compile-time errors in C. By using the following protocol, Alice proves Bob that she actually knows the secret Γ and that her watermark is present into the program, by revealing no knowledge to Bob about the watermark generation, nor the embedding, and nor the exact location of watermark in P. The protocol might be better understood by looking at Figure 5.1.

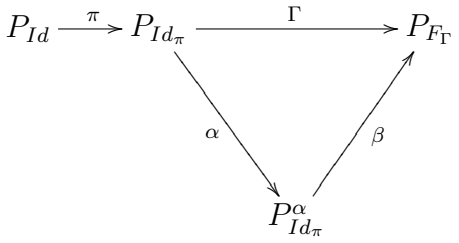


FIGURE 5.1: Watermark verification

5.3.4 Zero-knowledge Verification Protocol

☐ Alice generates the permutations α and β, and $\Gamma = \alpha \circ \beta$. She also computes $P^{\alpha}_{Id_\pi} = \alpha(P_{Id_\pi})$. Alice generates an ownership ticket (OT) with two commitments (for α and β) and a hash $H(P^{\alpha}_{Id_\pi})$. OT=$\langle C_1(\alpha), C_2(\beta), H(P^{\alpha}_{Id_\pi})\rangle$

Alice sends the signed OT to Bob.

☐ Bob flips a coin and ask Alice either

- to open commitment C_1, or
- to open commitment C_2

☐ Alice responds by opening either C_1 or C_2, based on Bob's request.

☐ Now the following two cases can arise on Bob's side:

- Case 1 (Alice opens commitment C_1): Bob computes $\alpha(P_{Id_\pi})$ from the knowledge of α contained in commitment C_1. Bob computes $H(\alpha(P_{Id_\pi}))$ and checks whether $H(P^{\alpha}_{Id_\pi}) = H(\alpha(P_{Id_\pi}))$.
- Case 2 (Alice opens commitment C_2): Bob computes $\beta^{-1}(P_{F_\Gamma})$ from the knowledge of β contained in commitment C_2. Bob computes $H(\beta^{-1}(P_{F_\Gamma}))$ and checks whether $H(P^{\alpha}_{Id_\pi}) = H(\beta^{-1}(P_{F_\Gamma}))$.

☐ Alice and Bob perform these steps repeated number of times (k). If all tests pass, Bob is convinced by Alice that the watermark is present in P_{Id_π}, and that Alice is the owner of P_{Id}.

Theorem 5.8. *The Zero-knowledge Verification Protocol is complete.*

Proof. During the protocol, no information about Γ is leaked. If an attacker can not determine Γ, then the value of α reveals nothing about the value of β, and vice versa, since S_m is a group. For every possible secret Γ and every possible revealed

permutation α, there exists one and only one β such that $\Gamma = \alpha o \beta$, and analogously this is true for a revealed β. Therefore, the *Zero-knowledge Verification Protocol* described above is complete. □

Theorem 5.9. *The Zero-knowledge Verification Protocol is weakly sound: after k iterations of the zero-knowledge protocol a cheating prover has a success probability* $1 - \sum_{i=1}^{k}(\frac{1}{2^i})$.

Proof. Suppose Bob wants his own watermark to appear in the program P_{Id}. Although he cannot change P_{Id}, he is free to add his own watermark W to P_{Id} and to scramble the program with his own hidden permutation Γ chosen from S_m. He pretends that his scrambled program is the true scrambling of P_{Id} induced by his own Γ. He may construct the ownership ticket by using either permutation α or β, by fooling a verifier in one of the two cases. Therefore, the probability of his success in each round is $\frac{1}{2}$. So after k iterations of the zero-knowledge protocol above, we get the resulting probability equals to $((((1 - \frac{1}{2}) - \frac{1}{2^2}))... - \frac{1}{2^k}) = 1 - \sum_{i=1}^{k}(\frac{1}{2^i})$. □

Example 5.1. *Let us illustrate our watermarking scheme by a simple program product.c in Figure 5.2 which performs multiplication operation by repeated addition method. LocId(P)= {a, b, sum, product, mul, x, y, prod, add, p, q, i, k, print_prod} and F(P)={mul(), add(), print_prod()}. The permutation group* S_{14} *consists of (14)!=87178291200 permutations and* S_3 *consists of (3)!= 6 permutations. Now suppose statement 1 of algorithm genW() in section 3.1 generates a hypothetical value r which is used to generate the secret permutation* $\pi_r \in S_{14}$. *Let* $\pi_r(LocId(P))$ *={product, prod, i, b, q, sum, a, mul, y, k, add, print_prod, x, p}. The next step is to perform the second hidden permutation* Γ *on the watermarked program* P_{π_r} *to generate its scrambled public version* P_{F_Γ} *by reordering the* $F(P)$. $P_{F_\Gamma} \in S_3$.

The original program P is shown in Figure 5.2 and corresponding watermarked program P_{π_r} *and the scrambled program* P_{F_Γ} *are shown in Figure 5.3, respectively. Notice that* P_{F_Γ} *generates compile time errors during compilation.*

5.4 Complexity

The computational complexity of watermark generation algorithm is linear. The watermark generation algorithm produces a permutation selected uniformly at

```
int add(int p, int q)
{
 int i,k=0;
 for(i=0;i<q;i++)
    k=k+p;
 return(k);
}

void mul(int x, int y)
{
 int prod;
 prod= add(x,y);
 print_prod(prod);
}

void print_prod(int product)
{
 printf("%d", product);
}

void main void()
{
 int a, b, sum;
 scanf("%d",&a);
 scanf("%d",&b);
 mul(a,b);
}
```

FIGURE 5.2: Original program

random from amongst all permutations in S_n. Let $r_{n-1}, r_{n-2}, ..., r_1, r_0$ be the sequence of random elements where $0 \leq r_i \leq i$. Since there are exactly $n \times (n-1) \times (n-2) \times ...2 \times 1 = n!$ such sequences, each difference sequence must produce a different permutation. Thus we should be able to unrank if we can take an integer r in the range $[0..n! - 1]$ and turn it into a unique sequence of values $r_{n-1}, r_{n-2}, ..., r_1, r_0$ where $0 \leq r_i \leq i$. Through the *Unrank* procedure this can be done in $O(n)$ operations [96].

In case of watermark embedding operation, if the original program has m statements and it contains n local variables, then the semantic preserving program transformation performs $O(m * n)$ operations to generate the watermarked program.

```
int y(int k, int add)
{
 int print_prod,x=0;
 for(print_prod=0; print_prod<add;
 print_prod++)
  x=x+k;
 return(x);
}

void q(int sum, int a)
{
 int mul;
 mul= y(sum,a);
 p(mul);
}

void p(int b)
{
 printf("%d", b);
}

void main (void)
{
 int product, prod, i, b;
 scanf("%d",&product);
 scanf("%d",&prod);
 q(product,prod);
}
```

```
void q(int sum, int a)
{
 int mul;
 mul= y(sum,a);
 p(mul);
}

int y(int k, int add)
{
 int print_prod,x=0;
 for(print_prod=0; print_prod<add;
 print_prod++)
  x=x+k;
 return(x);
}

void p(int b)
{
 printf("%d", b);
}

void main (void)
{
 int product, prod, i, b;
 scanf("%d",&product);
 scanf("%d",&prod);
 q(product,prod);
}
```

FIGURE 5.3: Watermarked and scrambled program

5.5 Security Considerations

In zero-knowledge watermark detection systems, *invertibility* or *ambiguity* attacks are very well known attacks where an adversary can create an ambiguous situation by deriving a forged watermark from a published work, and commits the forged watermark. Suppose Alice and Mallory use the same watermarking technique proposed here to watermark their softwares. And suppose Mallory wants his own watermark to appear in the program (P_{Id_π}) watermarked by Alice. Although he can't change P_{Id_π}, but he is free to add his own watermark $\pi(LocId(P_{Id_\pi}))$ to P_{Id_π} and scramble watermarked program P'_{Id_π} by his own hidden permutation Γ,

yielding $\Gamma(P'_{Id_\pi})$. And he pretends the owner of the program P. We can prevent such ambiguities by involving a third party *Trusted Time-Stamping Service*(TTSS) [67]. Before sending the signed OT to the *Verifier* (as described in section 3.4), the *Prover* must send it to a TTSS. The TTSS records the date and time the document was received and retains a copy of OT (signed by the *Prover*) for safe-keeping. The TTSS appends a signed *time-stamp* T for the submitter (i.e. the *Prover*) using his symmetric private key K_{TTSS} and send it back to the submitter (i.e. the *Prover*). So in our example the modified OT from Alice will look like, OT=$\langle C_1(\alpha), C_2(\beta), H(P^{\alpha}_{Id_\pi}), (T)_{K_{TTSS}}\rangle$. Then in the *time-stamp* verification phase, both the *Verifier* and the *Prover* have to come to TTSS and on the basis of their *time-stamp* values TTSS will solve the ambiguity about by revealing the actual owner of the software.

5.6 Watermarking and Fingerprinting

Traitors are dishonest buyers who redistribute the data to others without permission from the owner. The proposed watermarking scheme has a inherent traitor detection feature. The owner may use this watermarking scheme to embed a buyer-specific mark (permutations π and Γ) into his/ her code; he/she can subsequently detect the mark in pirated code and use the mark to identify the traitor who distributed the data. It is some how similar to fingerprinting where the owner embed a buyer-specific mark into a data copy provided to a buyer. Fingerprinting is a class of information hiding techniques that insert digital marks into data with the purpose of identifying the recipients who have been provided data, where Watermarking is another class of information hiding techniques whose purpose is to identify the sources of data. Both techniques can help protect data from piracy. And our algorithm has the provision to serve traitor detection (fingerprinting) as well. And this can be done by introducing identity certificate scheme like, Internet x.509 public key infrastructure certification in our protocol verification phase.

5.7 Software Watermarking: A Brief Survey

Research on software watermarking started in the 1990s. The patent by Davidson and Myhrvold [48] presented the first published software watermarking algorithm.

The early works on software watermarking include paper and patents, but the concepts in these works are preliminary and informal. For the first time, Collberg et al. presented detailed definitions for software watermarking [34]. Since then, several new software watermarking algorithms have been proposed. We can classify the most relevant existing software watermarking techniques as follows

- **Graph Based Software Watermarking:** Collberg and C. Thomborson [34] proposed a data Structure watermarking technique called Dynamic Graph Watermarking. The central idea is to embed a watermark in the topology of dynamically built graph structure. Code that builds this graph is inserted into the program to the watermarked. Because of pointer aliasing effects, graph-building code will be hard to analyze and detect, and it can be shown that it will be impervious to most de-watermarking attacks by code optimization and code ofucation. [30][32][125] enhance this idea. In Venkatesan et al. [127] proposed The software and the watermark code are converted into digraphs and new edges are introduced between the two by adding function calls between the software code and watermark code. Error-correcting capabilities were missing from the scheme and it was also susceptible to attacks that reorder the instructions and add new function calls. Instruction and block re-ordering attacks remain to be a problem for all these models.

- **Register Based Software Watermarking:** Watermark bits are encoded in the registers used for storing variables. Certain higher level language blocks by inline assembly code that controls which registers store which variables. Goal of the attack is to re-allocate variables in different registers to distort the encoding. Register-based software watermarking based on the QP algorithm (named after authors Qu and Potkonjak) [104][103] has been proposed in [95]. It changes the registers used to store variables depending on the variables required at the same time. The scheme is unable to survive register reallocation and recompilation attacks as well as secondary watermarking attacks. Also, inserting bogus methods renders the watermark useless by changing the interference graph.

- **Thread Based Software Watermarking:** Nagra and Thomborson [97] introduce a new dynamic technique for embedding robust software watermarks into a software program using thread contention. Multithreaded programs are inherently more difficult to analyze and the difficulty of analysis

increases with the number of threads that are live concurrently. The proposed technique embeds the watermark in the order and choice of threads which execute different parts of an application. The embedding is a two step process. Firstly, increasing the number of possible paths through the program by creating multiple threads of execution. The semantics of the old program are maintained by introducing locks. Secondly, other locks are added to ensure that only a small subset of the possible paths are in fact executed by the watermarked program. The particular paths that are executed encode the watermark. Proposed technique relies on introducing new threads into single threaded sections of a program. In an unsynchronized multithreaded program, two or more threads may try to read or write to the same area of memory or try to use resources simultaneously. This results in a race condition - a situation in which two or more threads or processes are reading or writing some shared data, and the final result depends on the timing of how the threads are scheduled.

- **Obfuscation Based Software Watermarking:** Code obfuscation is very similar to code optimization, except that with obfuscation one maximizes obscurity while minimizing execution time, whereas with optimization, just minimizes execution time. A potent defense against reverse engineering is obfuscation. By several transformations (lexical,control, data) on program body in [35] authors attempts to transform a program into an equivalent one that is harder to reverse engineer. Ertaul and Venkatesh [128] propose obfuscation techniques, based on composite functions, which are Array Index Transformation, Method Argument Transformation and Hiding Constants and an obfuscation algorithm based on Discrete Logs to Pack the Words and another one, based on Affine Ciphers, to Encode String Literals. Software protection tools like Sand Mark [33], Dot Obfuscator[46], JMangle [77], JObfuscator [78] and JHide [52] are all designed based on the principal theories of code obfuscation techniques.

- **Branch Based Software Watermarking:** Collberg et al. introduce path-based watermarking based on the dynamic branching behavior of programs which embeds the watermark in the runtime branch-ing structure of the program. The idea is based on the intuition that the forward branches executed by a program are an essential aspect of its computation and part of what makes the program unique. an obvious apparent drawback with using the

branch structure of a program to encode information is that, in principle, the branch structure of a program can be modified quite extensively without affecting program semantics, using well-known transformations such as basic block reordering, branch chaining (where the target of a branch instruction is itself a branch to some other location), loop unrolling, etc. This paper shows how error-correcting and tamper-proofing techniques can be used to make path-based watermarks resilient against a wide variety of attacks. Ginger Myles and Hongxia Jin [33] proposed a software watermarking scheme based on converting jump instructions or unconditional branch statements (UBSs) by calls to a fingerprint branch function (FBF) that computes the correct target address of the UBS as a function of the generated fingerprint and integrity check. If the program is tampered with, the fingerprint and integrity checks change and the target address will not be computed correctly. Unconditional branch statements (UBS) are converted to function calls to a special function called the branch function. The purpose of branch function is to transfer the control to the correct target address of the UBS [29].

- **Program Slicing Based Software Watermarking:** Software modules are split into open and hidden components. The open components are installed and executed on an un secure machine while the hidden components are installed and executed on a secure machine. While open components can be stolen, to obtain a fully functioning copy of the software, the hidden components, constructed by slicing the original software components, must be recovered. The hidden components are constructed in a manner that causes a great deal of effort to be required in finding the missing hidden components by observing the code of the open component and its runtime interactions with the hidden component and the recovery of hidden components constructed through slicing, in order to obtain a fully functioning copy of the software, is a complex task. Zhang and Gupta in [134] describe an algorithm that constructs hidden components by slicing the original software components.

- **Abstract Interpretation Based Software Watermarking:** A different approach of watermarking using static analysis in Abstract Interpretation frame work is presented in [128]. The basic idea is that the watermark is hidden in the program code in such a way that it can only be extracted by an abstract interpretation of the concrete semantics of this code. This static

analysis-based approach allows the watermark to be recovered even if only a small part of the program code is present and does not even need that code to be executed.The watermark is embedded in the values of some designated local variables during the program execution. The main advantage of this scheme is that the watermark can be recovered even if only small part of the code is available. The scheme can be attacked by obfuscating the program such that local variables representing the watermark cannot be located. Recently, Roberto Giacobazzi [59] shows how abstract interpretation more specifically, completeness provides an adequate developing a unifying theory of information hiding in software, by modeling observers (i.e., malicious host attackers) as suitable abstract interpreters. An observation can be any static or dynamic interpretation of programs intended to extract properties from its semantics and abstract interpretation provides the best framework to understand semantics at different levels of abstraction.This paper mainly covers completeness in the context of code obfuscation and watermarking.

- **Watermarking Systems:** There are 4 widely available watermarking systems for Java bytecode: Sandmark [33], Allatori [94], DashO [46]and jmark [94]. SandMark is a tool developed by Collberg et al. at the University of Arizona for research into software watermarking, tamper-proofing, and code obfuscation of Java bytecode. Sandmark contains 12 static software watermarking algorithms [28], which are implementations of some of the algorithms discussed in this paper. Allatori is a commercial Java obfuscator complete with a watermarking system created. DashO is a commercial Java security solution, including obfuscator, watermarking and encrypter. jmarks algorithm is also available in Sandmark. The static watermarking algorithms in all of these systems are susceptible to semantics-preserving transformation (distortive) attacks[68]. UWStego [31] is a tool, available on request, used for developing watermarking algorithms. Hydan [51], a system for steganographically embedding hidden messages in x86 assembly code, is available but is not aimed at watermarking and is therefore not resilient against attacks.

5.8 Conclusions

The proposed scheme has several benefits with respect to the current software watermarking techniques as it is robust against two typical software watermarking attacks.

- Before beginning to make modifications to the program, the attacker may make an attempt to gather some information about the watermark. A first step may be to try and determine if the program is in fact watermarked, and if it is, which watermarking system was used. Such types of attacks are called *collusive* attacks. The proposed watermarking scheme is invisible and does not reveals any information about the watermark and its location into the program, since the zero-knowledge proof is independent of the encoding and embedding, it is free from such attacks.

- Another very common software watermarking attack is the so called *program transformation* attacks, including *subtractive* attacks, *additive* attacks, *distortive* attacks etc. Also in this case, the proposed scheme performs well, since the watermark is spread in the entire program and the watermark detection depends on a secure collision resistant hash function.

- The scheme can be combined with a *time-stamp* technique to handle *invertibility/ambiguity* attacks.

This approach can be easily extended also to other programming languages by applying suitable permutation-based program transformations that exploit the particular programming languages' features.

Chapter 6

Conclusions

The thesis originated from the idea of extending Abstract Interpretation framework to some application fields, where both correctness and efficiency are crucial issues, and so semantics-based technologies based on abstractions may be the right tool to use. In particular the dissertation is focused on,

- Program slicing.
- Watermarking relational databases.
- Watermarking program source code.

Let us recall here the main contributions of the thesis as well as the corresponding research challenges that will deserve to be investigated in the future

1. Program slicing is used for reducing the size of programs to analyze. Nevertheless, sometimes this reduction is not sufficient for really improving the analysis. Suppose that some variables at some point of execution do not have a desired property. In order to understand where the error occurred it would be useful to find the statements affecting the property of these variables. Standard slicing may return too many statements, making it hard for the programmer to realize which ones caused the error. The proposed property driven program slicing technique allows to slice a program with respect to a given property, represented as an abstraction, instead of concrete values.

This kind of reasoning inherently relies on program semantics. Indeed, considering syntax alone is quite a good approximation in the case of concrete slicing, but becomes too imprecise when abstract properties are considered.

One direction of future work in this part, consists of accounting for more realistic frameworks. An inter-procedural formulation would be a first step in this direction. The logical and static analysis components needed to implement the algorithm deserve further study. The power of such techniques from the semantic point of view has to be investigated. Finally, effort will be put on implementing the presented framework.

2. Watermarking in relational frameworks is a relatively young technology that has begun its maturity cycle towards full deployment in industry-level applications. The proposed scheme shows a new way to see database watermarking in abstract interpretation based frame work. Proposed watermarking scheme is fragile in nature, it can detect and localized a single valued modifications. The scheme is distortion free and can be applied to any relational table irrespective to any attribute value.

   The solutions discussed in Chapter 4 need to be prototyped and validated on real data. The authentication proof of the database can be extended using zero-knowledge proof [63]. As in most zero-knowledge protocols, the proposed scheme requires many rounds of interactions between prover and verifier, which may not be efficient in practice. It is also not clear how to extend this scheme to watermarking relational databases. So it could be interesting to extend this work in this direction.

3. Recently, it has been shown how programs can be seen as abstractions of their semantics and how syntactic transformations can be specified as approximations of their semantic counterpart [59]. In particular, this result shows that abstract interpretation provides the right setting in which to formalize the relationship between code obfuscation and its effects on program semantics. In this setting, it would be interesting to see if our theoretical framework for software watermarking could be used to better understand and formalized the level of security that program diversity guarantees.

   Future work includes a full evaluation of the parametric framework on more realistic benchmark programs.

As for a formal conclusion we can say that, there is still lot of space for further research in applying semantics based analysis to various areas of software systems. Both on software manipulation, protection and database management the results that we presented in this thesis can be seen as very preliminary steps towards the development.

Most of the content of this thesis has already been published in proceedings of international conferences. *property driven program slicing* in [13, 36], *relational database watermarking* as *relational data table abstraction* in [14–16] and the *program source code watermarking* procedure based on *zero-knowledge proof system* in [17].

Bibliography

[1] Adelsbach, A., Katzenbeisser, S., and Sadeghi, A.-R. Watermark detection with zero-knowledge disclosure. *Multimedia Systems 9*, 3 (2003), 266–278.

[2] Agrawal, H. On slicing programs with jump statements. *In Proceedings of the ACM SIGPLAN94 Conference on Programming Language Design and Implementation 29*, 6 (1994), 302–312.

[3] Agrawal, H., and Horgan, J. Dynamic program slicing. *In Proceedings of the ACM SIGPLAN90 Conference on Programming Language Design and Implementation 25*, 6 (1990), 246–256.

[4] Agrawal, R., Haas, P. J., and Kiernan, J. Watermarking relational data: framework, algorithms and analysis. *The VLDB Journal 12*, 2 (2003), 157–169.

[5] Arnold, M., Schumucker, M., and Wolthusen, S. Techniques and applications of digital watermarking and content protection. *Artech House ISBN: 10: 1580531113* (2003).

[6] Badger, L., and Weiser, M. Minimizing communication for synchronizing parallel dataflow programs. *In International Conference on Parallel Processing (ICPP)* (1988), 122–126.

[7] Ball, T., and Horwitz, S. Slicing programs with arbitrary control-flow. *In Proceedings of the First International Workshop on Automated and Algorithmic Debugging, LNCS 749* (1993), 206–222.

[8] Barbara, D., Goel, R., and Jajodia, S. A checksum-based corruption detection techniques. *J.Comput. Security*, 11 (2003), 315–329.

[9] Barraclough, R. W., Binkley, D., Danicic, S., Harman, M., Hieron, R. M., Kiss, A., Laurence, M., and Ouarbya, L. A trajectory-based strict semantics for program slicing. *Elsevier 411*, 11-13 (2010), 1372–1386.

[10] Bassia, P., Pitas, L., and Nikolaidis, N. Robust audio watermarking in the time-domain. *IEEE Trans. Multimedia DOI: 10.1109/6046.923822* (2001), 232–242.

[11] Bergeretti, J., and Carre, B. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems 7*, 1 (1985), 37–61.

[12] Bertino, E., Ooi, B. C., Yang, Y., and Deng, R. H. Privacy and ownership preserving of outsourced medical data. *In Proceedings of the International Conference on Data Engineering* (2005), 521–532.

[13] Bhattacharya, S., and Cortesi, A. Property driven program slicing. *In the Proceeding of 20th Nordic Workshop on Programming Theory, NWPT 2008* (2008).

[14] Bhattacharya, S., and Cortesi, A. A distortion free watermarking framework for relational databases. *In the Proceeding of forth International Conference on Software and Data technology ICSOFT 2009, Sofia, Bulgaria* (2009), 229–234.

[15] Bhattacharya, S., and Cortesi, A. A generic distortion free watermarking technique for relational databases. *In the Proceeding of fifth International Conference on Information Systems Security, ICISS 2009, LNCS 5905* (2009), 252–264.

[16] Bhattacharya, S., and Cortesi, A. Database authentication by distortion free watermarking. *In the Proceeding of fifth International Conference on Software and Data technology, ICSOFT 2010 Athens, Greece (Best student paper award)* (2010), 219–226.

[17] Bhattacharya, S., and Cortesi, A. Zero-knowledge software watermarking for c programs. *In the Proceeding of International Conference on Advances in Communication, Network, and Computing CNC 2010, IEEE digital library (Selected for best paper award category)* (2010), 282–286.

[18] Binkley, D., Danicic, S., Gyimthy, T., Harman, M., Kiss, ., and Korel, B. A formalisation of the relationship between forms of program slicing. *Sci. Comput. Program 62*, 3 (2006), 228–252.

[19] Binkley, D., and Gallagher, K. Program slicing. *in Advances in Computers* (1996), 1–52.

[20] Binkley, D. W. Computing amorphous program slices using dependence graphs and data-flow model. *In ACM Symposium on Applied Computing ACM Press* (1999).

[21] Binkley, D. W., Danicic, S., Harman, M., Howroyd, J., and Ouarbya, L. A formal relationship between program slicing and partial evaluation. *Formal Aspects of Computing. SpimgerLink 18*, 2, 103–119.

[22] Canfora, G., Cimitile, A., and Luca, S. *Journal of information and Software Technology special Issue on Program Slicing, Elsevier*.

[23] Cartwright, I., and Felleisen, M. The semantics of program dependence. *In the Proceedings of PLDI* (1989).

[24] Cheng, J. Slicing concurrent programs a graph-theoretical approach. *In the Proceedings of the First International Workshop on Automated and Algorithmic Debugging, LNCS 749* (1993), 223–240.

[25] Choi, J., and Ferrante, J. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems 16*, 4 (July 1994), 1097–1113.

[26] Cimitile, A., Lucia, A. D., and Munro, M. Identifying reusable functions using specification driven program slicing: a case study. *Proceedings of International Conference on Software Maintenance, IEEE CS Press* (1995), 124–133.

[27] Cimitile, A., Lucia, A. D., and Munro, M. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance: Research and Practice 8*, 3 (1996), 145–178.

[28] Collberg, C. Sandmark algorithms. *Technical report, University of Arizona*, July (2002).

[29] Collberg, C., Carter, E., Debray, S., Huntwork, A., Linn, C., and Stepp, M. Dynamic path-based software watermarking. *In Proceeding of Conference on Programming Language Design and Implementation 39* (June 2004), 107–118.

[30] Collberg, C., Huntwork, A., Carter, E., and Townsend, G. Graph theoretic software watermarks: Implementation, analysis, and attacks. *Workshop on Information Hiding* (2004).

[31] Collberg, C., Jha, S., Thomko, D., and Wang, H. Uwstego.

[32] Collberg, C., Kobourov, S., Carter, E., and Thomborson, C. Error-correcting graphs for software watermarking. *In Proceedings of the 29th Workshop on Graph Theoretic Concepts in Computer Science* (2003), 156–167.

[33] Collberg, C., Myles, G., and Work, A. Sand mark a tool for software protection research. *IEEE Security and Privacy* (July/August 2003).

[34] Collberg, C., and Thomborson, C. Software watermarking: Models and dynamic embeddings. *In Proceeding of Principles of Programming Languages* (1999), 311–324.

[35] Collberg, C., and Thomborson, C. Watermarking,tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on Software Engineering 28*, 8 (August 2002), 735–746.

[36] Cortesi, A., and Bhattacharya, S. A framework for property driven program slicing. *1st Int. Conference on Computer, Communication, Control and Information Technology, Macmillan Publishers India Ldt ISBN/ISSN: 0230-063759-0* (2009), 118–122.

[37] Cortesi, A., and Halder, R. The dependence condition graph: Precise conditions for dependence between program points. *In Proceedings of the 10th International Workshop on Language Descriptions Tools and Applications (LDTA'10)* (2010).

[38] Cortesi, A., and Zanioli, M. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems and Structures 37*, 1 (2010), 24–42.

[39] Cousot, P. Abstract interpretation based formal methods and future challenges. *Logic and Comp 2000*, 2 (2000), 138–156.

[40] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Proceedings of the 4th ACM Symp. on Principles of Programming Languages* (1977), 238–252.

[41] Cousot, P., and Cousot, R. Constructive versions of tarskis fixed point theorem. *Pacific Math 82*, 1 (1979), 43–57.

[42] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. *In Proceedings of the 6th ACM Symp. on Principles of Programming Languages* (1979), 269–282.

[43] Cousot, P., and Cousot, R. Abstract interpretation frameworks. *Logic and Comp*, 2 (1992), 511–547.

[44] Cousot, P., and Cousot, R. Comparing the galois connection, and widening/narrowing approaches to abstract interpretation. *In Proceeding of PLILP* (1992).

[45] Cox, I., Kilian, J., Leighton, T., and Shamoon, T. A secure, robust watermark for multimedia. *LNCS 1174.* (1996), 317–333.

[46] DashO. http://www.preemptive.com/products/dasho/overview.

[47] Davidson, I., and Myhrvold, N. Method and system for generating and auditing a signature for a computer program. *Assignee:Microsoft Corporation.US Patent 5559884* (September 1996).

[48] Davidson, R., and Myhrvold, N. Method and system for generating and auditing a signature for a computer program. *US Patent 5559884* (1996).

[49] Duhamel, P., and Furon, T. An asymmetric public detection watermarking technique. *In Proceedings of the Third International Workshop on Information Hiding, LNCS 1768* (2000), 89–100.

[50] Eggers, J., Su, J., and B.Girod. Public key watermarking by eigenvectors of linear transforms. *In Proceedings of European Signal Processing Conference* (April 2000).

[51] El-Khalil, R. http://www.crazyboy.com/hydan/. *Hydan* (2004).

[52] Ertaul, L., and Venkatesh, S. Jhide a tool kit for code obfuscation. *In the Proceeding of 8th IASTED International Conference on Software Engineering and Applications* (Novenber 2004).

[53] Ettinger, R., and Verbaere, M. Untangling: A slice extraction refactoring. *In Proceedings of the 3rd International Conference on Aspect-Oriented Software Development* (2004), 93–101.

[54] Ferrante, J., Ottenstein, K., , and Warren, J. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*, 3 (1987), 319–349.

[55] Field, J., Ramalingam, G., and Tip, F. Parametric program slicing. *In Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (1995), 379–392.

[56] Gallagher, K., and Harman, M. Program slicing. *Information and Software Technology, special issue 40*, 11/12 (1998).

[57] Gallagher, K., and Lyle, J. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (1991).

[58] G.A.Venkatesh. The semantic approach to program slicing. *In Proceedings of the Conference on Programming Language Design and Implementation, SIGPLAN Notices 26*, 6 (1991), 107–119.

[59] Giacobazzi, R. Hiding information in completeness holes: New perspectives in code obfuscation and watermarking. *In the Proceedings of Sixth IEEE International Conference on Software Engineering and Formal Methods* (2008), 7–18.

[60] Giacobazzi, R., and Mastroeni, I. Abstract non-interference:parameterizing non-interference by abstract interpretation. *N. Jones and X. Leroy, editors, Proc. POPL 21*, 1 (2004).

[61] Giacobazzi, R., Ranzato, F., and Scozzari, F. Making abstract interpretations complete. *Journal of the ACM 47*, 2 (2000), 361–416.

[62] Gierz, G., Hofmann, K. H., Keimel, K., Lawson, J. D., Mislove, M., and Scott, D. D. Compendium on continuous lattices. *Springer Verlag* (1980).

[63] Goldwasser, S., Micali, S., and Rackoff, C. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing 18*, 1 (1989), 186–207.

[64] Gopal, R. Dynamic program slicing based on dependence relations. *In Proceedings of the Conference on Software Maintenance* (1991), 191–200.

[65] Gross-Amblard, D. Query-preserving watermarking of relational databases and xml documents. *In Proceedings of the Nineteenth ACM SIGMOD-SIGACTSIGART Symposium on Principles of Database Systems*, 191–201.

[66] Guo, H., Li, Y., Liu, A., and Jajodia, S. A fragile watermarking scheme for detecting malicious modifications of database relations. 1350–1378.

[67] Haber, S., and Stornetta, W. How to time-stamp a digital document. *Journal of Cryptology 3*, 2 (1991), 99–111.

[68] Hamilton, J., and Danicic, S. An evaluation of static java bytecode watermarking. *In Proceedings of the International Conference on Computer Science and Applications* (2010).

[69] Harman, M., and Danicic, S. Using program slicing to simplify testing. *Software Testing, Verification and Reliability 5*, 3 (1995), 143–162.

[70] Hartung, F., and Girod, B. Watermarking of uncompressed and compressed video. *Signal Processing 66*, DOI: 10.1016/S0165-1684(98)00011-5 (1998), 238–301.

[71] Hausler, P. Denotational program slicing. *In 22nd Annual Hawaii International Conference on System Sciences 11* (1989), 486–495.

[72] Hecht, M. Flow analysis of computer programs. *Elsevier* (1977).

[73] HMAC. The keyed-hash message authentication code. *FEDERAL INFORMATION PROCESS STANDARDS PUBLICATION* (2002).

[74] Hong, H., Lee, I., and Sokolsky, O. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. *In Proc. SCAM. IEEE* (2005).

[75] Horwitz, S., and Reps, T. The use of program dependence graphs in software engineering. *In Proceedings of the 14th International Conference on Software Engineering* (1992), 392–411.

[76] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–60.

[77] JMangle. http://www.elegant-software.com/software/jmangle/.

[78] Jobfuscator. http://download.com.com/3000-2417-10205637.html.

[79] Kamkar, M. An overview and comparative classification of program slicing techniques. *The Journal of Systems and Software 31* (1995), 197–214.

[80] Kiernan, J., and relational databases, R. A. W. *In Proceedings of the 28th International Conference on Very Large Databases VLDB* (2002).

[81] Kuck, D., Kuhn, R., Padua, D., Leasure, B., and Wolfe, M. Dependence graphs and compiler optimizations. *In Conference Record of the Eighth ACMSymposium on Principles of Programming Languages* (1981), 207–218.

[82] Lakhotia, A., and Deprez, J.-C. Restructuring programs by tucking statements into functions. *Information and Software Technology 40*, 11-12 (1998), 677–690.

[83] Langelaar, G., Setyawan, I., and Lagendijk, R. Watermarking digital image and video data: A state-of-art overview. *IEEE Signal Processing 17* (2000), 20–46.

[84] Lanubile, F., and Visaggio, G. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering 23*, 4 (1997), 246–459.

[85] Lemma, A., Aprea, J., and Kherkhof, L. A temporal-domain audio watermarking technique. *IEEE Trans. Signal Process. DOI: 10.1109/TSP.2003.809372* (2003), 1088–1097.

[86] Li, Y. *Handbook of database security, Springer Verlag*, 9780387485324 (2008), 329–355.

[87] Li, Y., Guo, H., and Jajodia, S. Tamper detection and localization for categorical data using fragile watermarks. *Digital Rights Management Workshop* (2004), 73–82.

[88] Main, A., and van Oorschot, P. Software protection and application security: Understanding the battleground. *International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography, Heverlee, Belgium* ([Available online]www.scs.carleton.ca/ paulv/papers/-softprot8a.ps 2003).

[89] Maruyama, K. Automated method-extraction refactoring by using block-based slicing. *In Proceedings of the 2001 Symposium on Software Reusability* (2001), 31–40.

[90] Mastroeni, I., and Nikolic, D. Abstract program slicing: From theory towards an implementation. *In Proceedings of the 12th International Conference on Formal Engineering Methods (ICFEM'10). LNCS 6467* (2010), 452–456.

[91] Mastroeni, I., and Zanardini, D. Data dependencies and program slicing: from syntax to abstract semantics. *Proceedings of ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (2008), 123–134.

[92] Meyers, T. M., and Binkley, D. Slice-based cohesion metrics and software intervention. *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)* (2004), 256–265.

[93] Millett, L., and Teitelbaum, T. Slicing promela and its applications to model checking. *In Proceedings on Model Checking of Software* (1998).

[94] Monden, A. jmark. *http://se.aist-nara.ac.jp/jmark/* (2003).

[95] Myles, G., and Collberg, C. Software watermarking through register allocation: Implementation, analysis, and attacks. *In the Proceedings of International Conference on Information Security and Cryptology, LNCS 2971* (2003), 274–293.

[96] Myrvold, W., and Ruskey, F. Ranking and unranking permutations in linear time. *Information Processing Letters* (October 2000).

[97] Nagra, J., and Thomborson, C. Threading software watermarks. *In Proceedings of 6th Information Hiding Workshop, LNCS 2971* (2004), 208–223.

[98] Ng, W., and Lau, H. L. Effective approaches for watermarking xml data. *DASFAA* (2005), 68–80.

[99] Nielson, F., Nielson, H., and Hankin, C. Principles of program analysis. *Springer Verlag* (1999).

[100] Ottenstein, K., and Ottenstein, L. The program dependence graph in a software development environment. *The Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments 19*, 5 (1984), 177–184.

[101] P. De Bra, J. P. An algorithm for horizontal decompositions. *Inf. Process. Lett 17*, 2 (1983), 91–95.

[102] Potdar, V., Han, S., and Chang, E. A survey of digital image watermarking techniques. *In the Proceeding of the 3rd International IEEE Conference on Industrial Informatics DOI: 10.1109/INDIN.2005.1560462* (August 2005), 709–716.

[103] Qu, G., and Potkonjak, M. Analysis of watermarking techniques for graph coloring problem. *In the Proceedings of International Conference on Computer Aided Design* (1998), 190–193.

[104] Qu, G., and Potkonjak, M. Hiding signatures in graph coloring solutions. *Information Hiding* (1999), 348–367.

[105] Quisquater, J.-J., Guillou, L. C., and Berson, T. A. How to explain zero-knowledge protocols to your children. *Advances in Cryptology 435* (1990), 628–631.

[106] Reps, T. Program analysis via graph reachability. *Information and Software Technology 40*, 11/12 (1998), 701–726.

[107] Reps, T., and Bricker, T. Illustrating interference in interfering versions of programs. *In Proceedings of the Second International Workshop on*

Software Configuration Management, ACMSIGSOFT Software Engineering Notes 17, 7 (1989), 46–55.

[108] Reps, T., and Yang, W. The semantics of program slicing and program integration. *In Proc. Colloq. on Current Issues in Programming Languages, LNCS 352* (1989).

[109] Rilling, J., and Klemola, T. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. *Proceedings of the 11th IEEE International Workshop on Program Comprehension* (2003), 115–121.

[110] Rilling, J., and Mudur, S. P. 3d visualization techniques to support slicing-based program comprehension. *Computers and Graphics 29*, 3 (2005), 311–329.

[111] Ruanaidh, Dowling, and Boland. Watermarking digital images for copyright protection. *IEEE ProcVision Signal, Image Procesing 143*, 4 (1996), 250–256.

[112] Sabelfeld, A., and Myers, A. Language-based informationflow security. *IEEE Journal on Selected Areas in Communications 21*, 1 (2003).

[113] Schneier, B. Applied cryptography. *John Wiley and Sons* (1996).

[114] Sion, R. Proving ownership over categorical data. *In Proceedings of the IEEE International Conference on Data Engineering ICDE* (2004).

[115] Sion, R. wmdb.*: A suite for database watermarking. *In Proceedings of the IEEE International Conference on Data Engineering ICDE* (2004).

[116] Sion, R., Atallah, M., and Prabhakar, S. On watermarking numeric sets. *In Proceedings of IWDW, LNCS* (2002).

[117] Sion, R., Atallah, M., and Prabhakar, S. Watermarking databases.

[118] Sion, R., Atallah, M., and Prabhakar, S. Rights protection for relational data. *In Proceedings of the ACM Special Interest Group on Management of Data Conference SIGMOD* (2003).

[119] Sion, R., Atallah, M., and Prabhakar, S. Relational data rights protection through watermarking. *IEEE Transactions on Knowledge and Data Engineering TKDE 16*, 6 (2004).

[120] Sion, R., Atallah, M., and Prabhakar, S. Ownership proofs for categorical data. *IEEE Transactions on Knowledge and Data Engineering TKDE* (2005).

[121] Snelting, G. C. Slicing and constant solving for validation of measurement software. *In Static Analysis Symposium, LNCS 1145* (1996), 332–348.

[122] Sukumarana, S., Sreenivasb, A., and Metta, R. The dependence condition graph: Precise conditions for dependence between program points. *Computer Languages, Systems and Structures 36*, 96-121 (2010), 577–581.

[123] Tanaka, K., Nakamura, Y., and Matsui, K. Ebedding secrect information in to a ditered multilevel image. *In Proceeding of IEEE military commun.* (1990), 216–220.

[124] the case for a new federal database protection law, D. P. M. http://www.siia.net/sharedcontent/gove/issues/ip/dbbrief.html. *SIIA:* (2000).

[125] Thomborson, C., Nagra, J., Somaraju, R., and He, C. Tamper-proofing software watermarks. *In the Proceeding of Conferences in research and practice in information technology* (2004).

[126] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages 3* (1995).

[127] Venkatesan, R., Vazirani, V., and Sinha, S. A graph theoretic approach to software watermarking. *In the Proceedings of 4th Information Hiding Workshop, LNCS 2137*, 1 (2001), 157–168.

[128] Venkatesh, S., and Ertaul, L. Novel obfuscation algorithms for software security. *Software Engineering Research and Practice* (2005), 209–215.

[129] Weiser, M. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. *PhD thesis, University of Michigan, Ann Arbor* (1979).

[130] Weiser, M. Programmers use slices when debugging. *Communications of the ACM 25*, 7 (1982), 446–452.

[131] Weiser, M. Reconstructing sequential behavior from parallel behavior projections. *Information Processing Letters 17*, 3 (1983), 129–135.

[132] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

[133] Wu, Y. Zero-distortion authentication watermarking. *ISC* (2003), 325–337.

[134] Zhang, X., and Gupta, R. Hiding program slices for software security. *In the Proceedings of First Annual IEEE/ACM International Symposium on Code Generation and Optimization* (March 2003), 325–336.