Department of Computer Science
The University of Auckland
New Zealand

Design and Evaluation of Software Obfuscations

Anirban Majumdar

2008

Supervisors:

Prof. Clark D. Thomborson

Dr. Stephen J. Drape

A thesis submitted in partial fulfilment of
the requirements of Doctor of Philosophy
in Computer Science

The University of Auckland

Thesis Consent Form

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

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

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

   Or

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

   (*Strike out 1 or 2*)

Signed: .....

Date: .....

Created: 5 July 2001

Last updated: 30 September 2007

Abstract

Software obfuscation is a protection technique for making code unintelligible to automated program comprehension and analysis tools. It works by performing semantic preserving transformations such that the difficulty of automatically extracting the computational logic out of code is increased. Obfuscating transforms in existing literature have been designed with the ambitious goal of being resilient against all possible reverse engineering attacks. Even though some of the constructions are based on intractable computational problems, we do not know, in practice, how to generate hard instances of obfuscated problems such that all forms of program analyses would fail.

In this thesis, we address the problem of software protection by developing a weaker notion of obfuscation under which it is not required to guarantee an absolute blackbox security. Using this notion, we develop provably-correct obfuscating transforms using dependencies existing within program structures and indeterminacies in communication characteristics between programs in a distributed computing environment. We show how several well known static analysis tools can be used for reverse engineering obfuscating transforms that derive resilience from computationally hard problems. In particular, we restrict ourselves to one common and potent static analysis tool, the static slicer, and use it as our attack tool. We show the use of derived software engineering metrics to indicate the degree of success or failure of a slicer attack on a piece of obfuscated code. We address the issue of proving correctness of obfuscating transforms by adapting existing proof techniques for functional program refinement and communicating sequential processes.

The results of this thesis could be used for future work in two ways: first, future researchers may extend our proposed techniques to design obfuscations using a wider range of dependencies that exist between dynamic program structures. Our restricted attack model using one static analysis tool can also be relaxed and obfuscations capable of withstanding a broader class of static and dynamic analysis attacks could be developed based on the same principles. Secondly, our obfuscatory strength evaluation techniques could guide anti-malware researchers in the development of tools to detect obfuscated strains of polymorphic viruses.

Acknowledgements

This thesis has benefitted immensely from the wisdom and advice of many individuals. This is a good opportunity to thank them all.

First and foremost, I am deeply indebted to my supervisors Professor Clark Thomborson and Dr. Stephen Drape for mentoring me towards a completion in the minimum allowable time. Over the last three years, Professor Thomborson has given me both the freedom to pursue my own research interests and the insightful guidance that I have needed. Through him I learnt the art of lateral thinking and the use of rigour in scientific research. I am grateful to him for funding my research through the New Zealand government's New Economy Research Foundation fund on "Securing Software for New Zealand". I am extremely fortunate to have had Dr. Drape as my co-supervisor over the period of last eighteen months. Dr. Drape has not only been an excellent thesis advisor but also a valued friend. During the troubled times of funding uncertainty, he has taught me to persevere and remain focussed on my research. Our mutual admiration for progressive rock and Ricky Gervais resulted in some memorable time spent at rock concerts and in front screens appreciating countless episodes of The Office. Dr. Drape has been instrumental in introducing me to programming language theory and I am particularly indebted for his help in adapting a framework for proving correctness of imperative transforms. I am thankful to Professor Thomborson for his help in shaping up the final thesis draft and to Dr. Drape for taking time off of his busy schedule at Oxford University to proofread my thesis drafts.

I have benefitted immensely from interactions with the former members of the Secure Systems Group. I grateful to Dr. Antoine Monsifrot, of Thomson R & D France, for his collaborative efforts on static analysis experiments with BDDBDDB and Indus. Dr. Jasvir Nagra, of University of Trento, has been my peer mentor all these years and have enlightened me with his mastery of two dozen programming languages and analytical skills. Michael Stay, of Google Mountain View Labs, fascinated me with his alternative view of distributed opaque predicates using the cryptographic theory of multi-party computations. I have thoroughly enjoyed the intriguing discussions on computer forensics and technorealism with Dr. Simson Garfinkel of Harvard University. I am thankful to my former mentors Professor Hon Fung Li of Concordia University, Canada, Professor Subhansu Bandopadhyay and Dr. Nabendu Chaki of Calcutta University, India for their unvarying impetus that catalysed my yearning to take up research in computer science as my professional career. I am thankful to my school and university friends in India and abroad for providing me with encouragement and entertainment right from the beginning. I am thankful to Dr. David Pearce, my oral examiner, for his help in improving the overall quality of the thesis with his detailed suggestions.

My greatest gratitude goes to my parents for their unconditional support during the course of my PhD. My wonderful Ma has nurtured me with love and care all my life.

She has been the source of limitless encouragement during these difficult years. My dad motivated my inquisitiveness from very early on and supported my career decisions no matter where they took me. Being a scientist himself, my dad provided immensely valuable suggestions at critical junctures of my research and thesis writing. I am thankful to my sister for helping me choose a career in computer science over medicine right after my high school days. I thank Sikharini Majumdar for her encouraging phone conversations and for generously sharing the plight of fellow PhD students across the Pacific. She made me feel that I am not alone, after all. Finally, I am thankful to my wife Raka; her unwavering patience and encouragement during these trying times have made it all worth it. I could not have envisioned this personal journey without her. It is to Raka that I dedicate this thesis.

Contents

List of Figures

List of Tables

1

Introduction

No less than five PhD theses [6, 132, 136, 50, 36], three Masters dissertations [77, 131, 8], tens of publications, and at least two patents [124, 117] have appeared on the issue of software protection by means of code obfuscation since the 1997 publication of a technical report by Collberg *et al.* [27]. The motivation for protecting software through obfuscation arises from the problem of software piracy, which can be summarised as a reverse engineering process undertaken by an adversary when stealing intellectual artefacts (such as a patented algorithm) from a commercially valuable software with the intention of making illegal derivative copies or tampering with DRM routines in order to bypass license authentication checks [93]. The 2005 annual Global Software Piracy Report [1] from Business Software Alliance (BSA) stated that "35% percent of the packaged software installed on personal computers (PC) worldwide

in 2005 was illegal, amounting to $34 billion in global losses due to software piracy". This is one of the primary reasons why commercially popular software such as the Skype internet telephony client [14], the SDC Java DRM [112], and most license-control systems rely, at least in part, on obfuscation for their security. With the advent of platform independent mobile code, malicious reverse engineering has become a serious challenge to the software industry.

The goal of software protection through code obfuscation is to transform the source code of an application to the point that it becomes unintelligible to automated program comprehension tools or becomes unanalysable to a human adversary interpreting the output of program analyses run on the obfuscated application. Depending on the size of software and the complexity of transforms, a human adversary may find the obfuscated code too difficult to reverse engineer. However, as Thomborson *et al.* [122] noted, software that is simple and manageable enough to be completely analysed by human adversaries could presumably be redeveloped from scratch by attackers at reasonable cost. Therefore, the justification of applying complicated and costly obfuscating transforms lies in protecting complex programs that are otherwise untraceable manually by human adversaries.

Barak, in his informal discussion on the limitations of obfuscation [5], discussed two types of security objectives: *provable* and *fuzzy security*. He defined provable security as a concept defined in terms of an instance of a hard problem; i.e., there exists a mathematical proof which shows that if someone can break a particular implementation, then there is an efficient algorithm for a well known hard computational problem. Fuzzy security, on the other hand, is ad-hoc in nature and lacks a rigorous definition. A fuzzily secure security technique claims to provide security but the claims are unproven since there is no underlying theoretical basis. Nowadays, for most cryptographic tasks, we do not need to use fuzzy security since we have well defined security definitions and constructions in some well defined model. The assumptions of the model are, typically, that a user can reveal a password to a system on some channel that is secure from eavesdropping; and that the system can manipulate this password, and its associated security key in some "black box" computational environment that is secure from eavesdroppers. One of the

goals of research into obfuscation is to describe ways in which black boxes, and secure I/O with users, can be efficiently and inexpensively implemented. Current technology for implementing cryptographic systems relies on specialised tamperproof hardware, which is, of course only truly secure against a limited range of adversaries even if it is used properly. Unfortunately with obfuscation, so far no one has been able to come up with a realistic model which would support provable obfuscation. Furthermore, the absence of a provable security model for obfuscation prevents us from making claims on the resilience of obfuscation techniques.

In the next section we highlight some alternative technologies for software protection. Following this, we give the motivation and outline the structure of the thesis.

1.1 Alternative protection techniques

A couple of orthogonal software protection techniques exist in the security literature — the first uses *guard* modules and encryption routines to secure a piece of running code and the other uses *remote entrusting* techniques.

Aucsmith [4] showed how *guard* modules and encryption routines could be used to restrict the tampering of software by observation and modification. His guard modules were called Integrity Verification Kernels (IVR). Aucsmith's IVRs operated in interleaved fashion and were responsible for runtime verification of code integrity. The IVRs used encryption routines to interleave tasks and distribute secrets throughout the code and *interlocked trust* by binding with other IVRs. Following Aucsmith's proposal, similar ideas of using specialised guard modules for tamperproofing software began to emerge. Two prominent results were published by Chang *et al.* [19] and Horne *et al.* [57]. These two similar proposals were presented at the same conference (Digital Rights Management 2001). Chang *et al.* [19] pointed out that Aucsmith's technique has a disadvantageous cost associated with encrypting and decrypting a window of security-sensitive program instructions before and after each execution round. Also, Aucsmith's decryption routine has to be embedded within the ported code and the problem of protecting this routine

is analogous to the original problem of protecting the software itself. Chang's improved solution used guards that checksummed code for verifying integrity. Resilience was argued on the grounds that interlocked guard modules would be hard to deactivate because each of Chang's guards would check the integrity of other guards.

The other approach of software protection promotes the use of *remote entrusting* techniques [139, 16]. *Server-side execution* is an example of software protection where the software developer only makes a set of services available to clients while the core of the software executes at his/her site (online games for example). Zhang *et al.* [139] used slicing to extract security-sensitive code and store it on a hardware dongle. Ceccato *et al.* [16] improved on the efficiency by using barrier slicing which resulted in smaller slices and lesser communication overhead between the server and the client. Two main disadvantages could be summarised as the need for an always-on network connection and the delay incurred due to barrier synchronisations between the client and the server.

1.2 Motivation

When the idea of research in designing obfuscations was conceived four years ago, there were two distinct emerging trends. The first followed a "cryptographic approach" and the second followed a "compilers' approach". The main proponent of the first approach was Ehud Barak who in his PhD thesis [6], and with co-authors in a landmark article [7] established that every obfuscator will fail to completely obfuscate some programs and that no obfuscator can guarantee a perfect blackbox. The second approach was spearheaded by Christian Collberg *et al.* in a heavily referenced technical report [27]. Collberg *et al.* proposed much relaxed requirements for obfuscation (lacking theoretical basis) under the assumption that a determined adversary will eventually be able to deobfuscate any program. There was very little commonality between these two trends and it is imprudent to overlap the security goals for these two different approaches together. For instance, it is ambitious to impose Barak's stringent blackbox security goal on programs that have been obfuscated with the compiler-like transforms proposed by Collberg *et al.* Secondly,

even for transforms which rests on intractability results of computationally hard problems (such as Chenxi Wang's approach in [132]) suffer from one fundamental drawback — we do not know in practise how to arbitrarily generate instances of NP-complete problems.

In general, we believe that it is impossible to come up with one obfuscating transform capable of withstanding all possible deobfuscation attacks. Similarly, it is also ambitious to try designing a general purpose attack tool for "cracking" all possible obfuscations. We will investigate these two issues in this thesis. Our motivation is to develop a much weaker non-cryptographic notion of obfuscation that will provide "greybox" security within a restricted attack model. Our obfuscations will be designed both using the inherent dependencies existing within program constructs as proposed in [27] and using the communication induced dependencies between programs executing in distributed computing environment. Thus our *design parts* will have compliers as well as distributed computing flavour to them. Since the security requirement is also relaxed, we do not need to guarantee an absolute (0/1) cryptographic security for our obfuscations in a restricted attack model. We will design metrics for evaluating the obfuscatory strength of our transforms and thus the *evaluation part* will have a software engineering feel to it. We will also provide theoretical basis for our obfuscations using the principles laid out in [36] and [113]. Our *correctness proofs* will have program refinement and Communicating Sequential Process (CSP) flavours to them. The common binding thread is the ways in which we will use standard intra-program dependencies (such as control- and data-flow) and inter-program dependencies (such as message-passing and distributed global state) for designing our obfuscations. Our obfuscations and evaluation strategies are a step forward for pursuing research for practical software protection, the goals of which are not as stringent as cryptographic security.

It is worth noting that we treat obfuscation as a tool for software protection in the thesis. Therefore, we will assume that our adversaries will try to attack code in a way so that they can understand and identify the portions of code that interest them (license control, for instance). A complementary approach is to treat obfuscation as a "misuse" case used by virus writers to evade detection by anti-malware software [23, 24, 106].

Indeed, we believe that anti-virus developers could benefit from techniques for obfuscatory strength evaluation but we do not treat polymorphic virus development as an application for our obfuscations in this thesis.

1.3 Outline of the thesis

We start off with two definitions of obfuscation proposed by Barak and Collberg *et al.* respectively in Chapter 2. In Chapter 2 we also give a background of several obfuscations existing in the literature which claim their resilience from computationally intractable problems. In Chapter 3, we discuss techniques to statically evaluate the obfuscatory strength of transforms and identify one particular technique, called static slicing, as our attack tool for the rest of the thesis. We then show, in Chapter 4, how to use dependencies within programs (intra-process dependencies) to design obfuscations for deterring slicing attacks. We call these obfuscations "slicing obfuscations". We also evaluate obfuscatory strength of slicing obfuscations using derived slicing metrics in this chapter. Chapter 5 deals with the correctness issues of slicing obfuscations using program refinement techniques . In Chapter 6 we use communication-induced dependencies in programs executing in distributed computing environments (inter-process dependencies) for designing obfuscations. In particular, we design a control-flow obfuscation, called "distributed opaque predicate" and illustrate its use with an example in Chapter 6. We give a correctness proof sketch for distributed opaque predicates and a slicer experiment to evaluate its obfuscatory strength in Chapter 7. We finally conclude the thesis with linking up these two approaches for designing obfuscations and comment on extensions that can be made on the work contained herein.

2

Obfuscation and Related Work

In this chapter, we provide a thorough background of obfuscation and different obfuscating transforms that have been existing in the literature till date and identify some of the issues with them. We start by providing different definitions of obfuscation that have been proposed to date. Next we provide a brief taxonomy of obfuscation from Collberg *et al.* [27] and highlight the different categories of obfuscation. We give an in depth account of data and control-flow obfuscations later in the chapter. For control-flow obfuscations we focus on two particular transforms that rely on computationally intractable problems for their resilience and discuss several salient features about them. This chapter serves as the basis for evaluation of obfuscatory strength of transforms which will be discussed in Chapter 3.

2.1 Defining Obfuscation

The first formal definition of obfuscation was given by Collberg *et al.* [27, 28]. They defined an obfuscator in terms of a semantic-preserving transformation function $\mathcal{T}$ which maps a program $\mathcal{P}$ to a program $\mathcal{P}'$ such that if $\mathcal{P}$ fails to terminate or terminates with an error, then $\mathcal{P}'$ may or may not terminate. Otherwise, $\mathcal{P}'$ must terminate and produce the same output as $\mathcal{P}$.

They introduced the notion of *potency of obfuscation* i.e., potency of $\mathcal{T}$, as:

$$\mathcal{T}_{pot}(\mathcal{P}) = \frac{E(\mathcal{P}')}{E(\mathcal{P})} - 1$$

where $E(\mathcal{P})$ is the complexity of program $\mathcal{P}$ defined in terms of qualitative metrics, such as:

- Cyclomatic Complexity - the complexity of a function increases proportionately with the number of predicates in the function [88].
- Nesting Complexity - the complexity of a function increases proportionately with the nesting level of conditionals in the function [51].
- Data structure Complexity - the complexity increases with the complexity of static data structures declared in the program (for example, with array dimension) [92].

$\mathcal{T}$ is said to be a *potent obfuscating transform* if $\mathcal{T}_{pot}(\mathcal{P}) > 0$. Note that a potent transform $\mathcal{T}$ may result in an inefficient obfuscated program $\mathcal{P}'$. Additionally, Collberg *et al.* defined three additional desirable quality metrics on obfuscation:

Resilience How difficult $\mathcal{T}$ is for a deobfuscator to undo. The resilience of a transformation $\mathcal{T}$ is a combination of *programmer effort* (the amount of time required to construct an automatic deobfuscator which is capable of effectively reducing the potency of $\mathcal{T}$) and *deobfuscation effort* (the execution time and space required by such an automatic deobfuscator to effectively reduce the potency of $\mathcal{T}$).

Stealth How well the code introduced by $\mathcal{T}$ blends with the code of $\mathcal{P}$. It is a context-sensitive metric. Code introduced by $\mathcal{T}$ may be stealthy in one program but unstealthy in another.

Cost How much computational overhead (time/space penalty) $\mathcal{T}$ adds to $\mathcal{P}'$.

Barak *et al.* [7] strengthened the formalism of Collberg *et al.* by defining an obfuscator, $\mathcal{O}$, in terms of a *compiler* that takes a program, $\mathcal{P}$, as input and produces an obfuscated program, $\mathcal{O}(\mathcal{P})$, as output such that the following three conditions hold:

1. *Functionality:* The obfuscated program, $\mathcal{O}(\mathcal{P})$, should have the same functionality (i.e., input/output behavior) as the input program $\mathcal{P}$. The functionality requirement states that the input program and the obfuscated program should compute the same function.

2. *Efficiency:* The obfuscated program must not be much less efficient than the input program, i.e., the description length and running time $\mathcal{O}(\mathcal{P})$ are at most polynomially larger than that of $\mathcal{P}$.

3. *Virtual-black box simulation:* Anything that can be efficiently computed from $\mathcal{O}(\mathcal{P})$ can be efficiently computed given oracle access to $\mathcal{P}$.

Thinking in terms of a *virtual black-box*, an obfuscation function is a *failure* if there exists at least one program that cannot be completely obfuscated by this function, that is, if an adversary could learn something from an examination of the obfuscated version of this program that cannot be learned (in roughly the same amount of time) by merely executing this program repeatedly. Using this notion of *virtual black-box*, Barak *et al.* [7] established that every obfuscator will fail to completely obfuscate some programs . Drape observed in his DPhil thesis [36], that the "Virtual Black Box" property is too strong. Obfuscators will be of practical use even if they do not provide perfect black-boxes. Therefore, the focus of obfuscation research has shifted to finding obfuscating transforms that are *difficult* (but not necessarily *impossible*) for an adversary to reverse engineer. The goal of such research

is to find obfuscating transforms such that the resources required for undoing them are too expensive to be worth the while of adversaries. Collberg *et al.* classified obfuscating transforms into three useful categories:

- **Layout obfuscation:** This category of transforms changes or removes useful information from the intermediate language code or the source code without affecting the instructions that contribute to actual computation. Usually removing debugging information, comments, and scrambling/renaming identifiers fall within the domain of layout obfuscation. Most of the popular commercial obfuscators such as Dotfuscator [62], DashO [61], Zeliz [138], and Proguard [107] use trivial layout transformation techniques.

- **Data obfuscation:** This category of transforms targets data and data structures contained in the program. Using these transformations, data encoding can be changed, variables can be split or merged, and arrays can be split, folded, and merged.

- **Control-flow obfuscation:** The objective of this category of transforms is to alter the flow of control within the code. Reordering of statements, methods, loops and hiding the actual control flow behind irrelevant conditional statements classify as control-flow obfuscation transforms.

In the following sections, we discuss two categories of obfuscations which we use in the thesis — namely, data and control-flow. Data obfuscations have been extensively used in Chapter 4 and 5. For discussing control-flow obfuscations, we stress on those transforms which derive their obfuscatory resilience from complexity results of computationally intractable problems. Chapter 6 of the thesis concerns control-flow obfuscations.

2.2 Data obfuscations

The objective of data obfuscations is to transform the data structures in the source code into a form which is difficult for an adversary to comprehend [27]. Collberg *et al.* classi-

fied these transformations into four distinct categories depending on the way they affect storage, encoding, aggregation, and ordering of data. In the next few subsections, we elucidate data obfuscations from [27] which we will frequently refer to in Chapters 4 and 5 of the thesis.

2.2.1 Variable encoding

Encoding obfuscations chooses "unnatural" encoding transforms for common data structures. Since variable encodings can be used for compiler optimisations, we need to choose unnatural transforms for obfuscations. A simple encoding example is proposed where an integer variable i is replaced with

$$i' = c_1.i + c_2$$

where c_1 and c_2 are constants. A simple code snippet using this for of encoding where c_1 is chosen to be a power of 2 is given below (after [27]):

```
int i = 1;                          int i = 11;
while (i < 1000) {                  while (i < 8003) {
  ... A[i] ... ;          ⇝           ... A[(i − 3)/8] ... ;
  i + +;                              i+ = 8;
}                                   }
```

It is worth noting that overflow issues when encoding floating point variables need to be taken care of. Collberg *et al.* comments that such simple encodings are not potent and can be easily reverse engineered using standard compiler analysis techniques. It is therefore challenging to use such simple obfuscations to defeat any of the available program analysis techniques.

2.2.2 Variable splitting and merging

Variable splitting and merging data obfuscations are similar to the encoding obfuscation. In splitting, a variable V of restricted range can be split into two or more variables

$p_1, \ldots p_k$. Collberg *et al.* observed that both the potency and the cost of this transformation will grow with k. They also noted that for a variable V of type T to be split into two variables p and q of type U, the following three pieces of information need to be provided:

- a function $f(p, q)$ which maps the values of p and q into V.
- a function $g(V)$ which maps the value of V into p and q.
- new operations cast in terms of operations on p and q (corresponding to the primitive operations on values of type T).

Similarly for merging variables, two or more scalar variables (can hold only one value at a time) $V_1 \ldots V_k$ can be merged into one variable V_M if the combined range of $V_1 \ldots V_k$ fits within the precision of V_M. As a simple example, two 32-bit integer variables X and Y can be merged into a 64-bit variable Z as follows (after [27]):

$$Z(X, Y) = 2^{32}.Y + X$$

Collberg *et al.* observes that the resilience of merging obfuscations is low since a deobfuscator only needs to find out the set of arithmetic operations that are being applied to a particular variable in order to detect that it is composed of merged variables.

2.2.3 Array transformations

Collberg *et al.* extended the concept of splitting and merging of scalar variables to composite data structures such as arrays. Arrays can be *split* into several sub-arrays, two or more arrays can be *merged* into one bigger array, *folded* so as to increase the number of dimensions, or *flattened* to decrease the number of dimensions. The self-explanatory Figure 2.1 from [27] explains each of these obfuscations in sequence.

Drape in his DPhil thesis [36] dealt with modeling data obfuscations for imperative languages using functional programming concepts. He observed that while obfuscating programs, two different approaches can be taken — either the algorithm or/and the

```
int A[9];                                    int A1[4], A2[4];
A[i] = ...;                                  if ((i%2) == 0)A1[i/2] = ...;
  ...                                          A2[i/2] = ...;
  ...                                          ...
int B[9], C[19];                             int BC[29];
B[i] = ...;                                  BC[3 * i] = ...;
C[i] = ...;                                  BC[i/2 * 3 + 1 + i%2] = ...;
  ...                                          ...
int D[9];                                    int D1[1, 4];
for (i = 0; i <= 8; i++)          ⇝          for (j = 0; j <= 1; j++)
  D[i] = 2 * D[i + 1];                         for (k = 0; k <= 4; k++)
  ...                                            if (k == 4)
  ...                                              D1[j, k] = 2 * D1[j + 1, 0];
  ...                                            else
  ...                                              D1[j, k] = 2 * D1[j, k + 1];
int E[2, 2];                                   ...
for (i = 0; i <= 2; i++)                     int E1[8];
  for (j = 0; i <= 2; i++)                   for (i = 0; i <= 8; i++)
    swap(E[i, j], E[j, i]);                    swap(E[i], E[3 * (i%3) + i/3]);
```

Figure 2.1: Example of array transformations (after [27])

data types can be obfuscated. He took the second approach and defined obfuscations on abstract data types using the functional program refinement. Obfuscating abstract data types instead of concrete implementations provides a simpler way of proving correctness of obfuscation. Moreover, obfuscating data types gives an extra level to add obfuscations since we are not constrained to think about implementation intricacies. He also generalised some of the data obfuscations and in particular the array split transformations. The details of these generalisations can be found in [37].

2.3 Control-flow obfuscations

As stated earlier, the objective of control-flow obfuscation is to alter the control flow of code such that an attacker is falsely led into believing that the program execution takes an obfuscated false control flow. We now outline a few control-flow obfuscation techniques using the taxonomy of [27]:

Aggregation/De-aggregation The original control-flow logic is disturbed by coalescing unrelated methods or splitting related methods. The DOJ (Design Obfuscator

for Java) [116] uses this type of transform to perform control-flow obfuscation. Additionally, method inlining, outlining, cloning, and loop transformations are also categorised as aggregation type control-flow obfuscation.

Ordering This category performs reordering operations on statements, loops, and expressions to disturb the locality of related information.

Spurious computations This type of obfuscation is done by modifying the real control flow by adding spurious computation blocks. Opaque predicates — which we will describe fully in 2.3.2 — play an important role in performing semantic-preserving control-flow obfuscation by adding spurious code to conditional constructs in the program.

Most of the control-flow obfuscating transforms derive their hardness from the intractability results of pointer aliasing. Aliasing occurs when two variables refer to the same memory location and it is well-known in compiler theory that precise and flow-sensitive alias analysis is undecidable in languages with dynamic allocation, loops, and if-statements [73, 108, 17] . Several authors have used this intractability result as a theoretical basis for designing their control-flow obfuscating transforms. Most notable among these are:

- Obfuscations such as identifier renaming/overloading [124], class coalescing and class splitting [116] rely on the difficulty that in the presence of method-overloading, precisely determining if there exists an execution path in a program for which a given reference points to a given method at a point of the program execution [110, 99].

- Opaque predicates rely on difficulty of the alias analysis problem and the shortcomings of the available conservative algorithms for analysing the control-flow of programs [27, 28].

- Control-flow flattening techniques rely on the `NP-hardness` of precisely determining the indirect branch target addresses of dispatchers in the presence of aliased pointers [133].

In the next few subsections, we summarise the salient aspect of a few of these intractable obfuscating transforms.

2.3.1 The dynamic dispatcher model of control-flow obfuscation

Wang [133] and Chow *et al.* [22] made the first commendable attempt to lay theoretical foundations for control-flow obfuscation of sequential programs. The theoretical basis in Wang's technique is the `NP-complete` [45] argument of determining precise indirect branch target addresses of dispatchers in the presence of aliased pointers and that for Chow *et al.* on the `PSPACE-complete` [45] problem of determining the reachability of a flattened program dispatcher. Because we do not have an average-case hard instance generator for NP-complete or PSPACE-complete problems, these theoretical foundations are still incomplete. However even in their current state, these models provide some "fuzzy" confidence in the security of the real-world obfuscation system they describe. Eventually we would hope to prove security results in some model of obfuscation. Then the only real-world attacks would be subversive of the model assumptions, analogous to how a provably-secure cryptographic system can never be "cracked", but may still be subverted e.g. by "social engineering" methods of password discovery.

Control-flow flattening makes all basic blocks appear to have the same set of predecessors and successors. The actual control-flow during execution is determined dynamically by a dispatcher. Wang suggested a three step control-flow flattening strategy as briefly described below (adapted from [125]):

Basic Control-flow flattening Each basic block in the program is flattened so that they have the same set of predecessors and successors. Then a dispatcher variable is determined and at runtime, each basic block assigns to this dispatcher variable a value indicating which basic block is to be executed next. A `switch` block uses the dispatcher variable to jump indirectly through a table to the next control-flow block. Figure 2.2 is a program and its corresponding control-flow graph. Upon the application of basic control-flow flattening, we get the control-flow graph of Figure

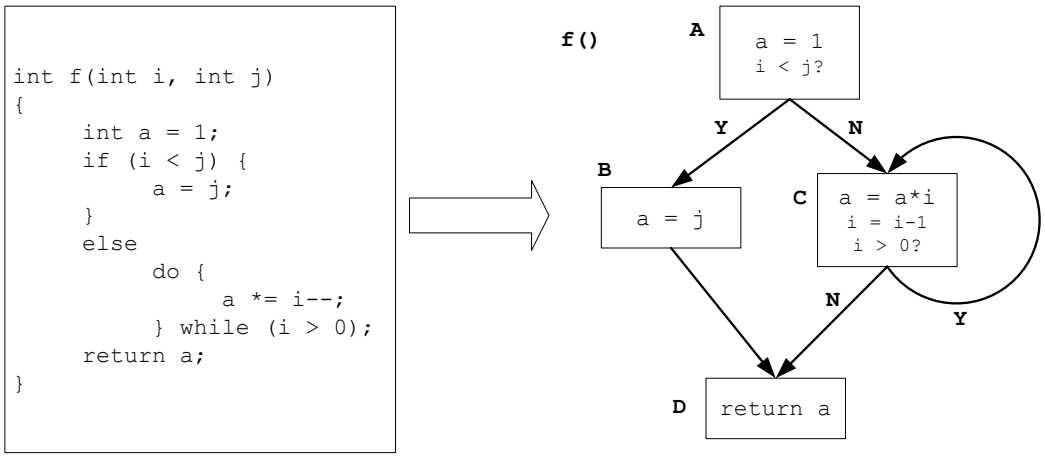

Figure 2.2: The example program and its control-flow graph (after [125])

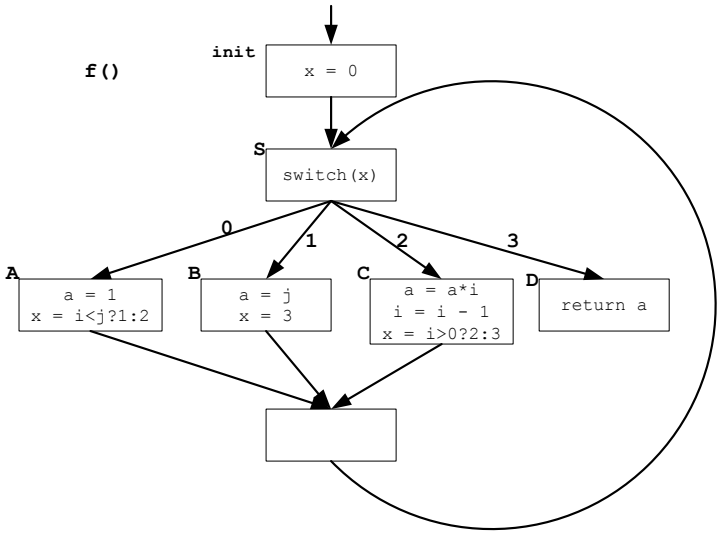

Figure 2.3: Control-flow graph after basic control-flow flattening (after [125])

2.3. `S` is the `switch` block and `x` is the dispatcher variable that routes control to block `A` which is the original entry point of function `f()`. After this initial entry, control-flow is is guided by assignments to variable `x` in other basic blocks.

Adding inter-procedural data flow Basic control-flow flattening allows intra-procedural analysis of the original control-flow since the values assigned to the dispatcher variable are available within the function itself. The resilience can be improved by inter-procedural value passing. The authors proposed using a global array to pass the dispatch variable values. At each call site to the function, these values are written into the global array starring at some random offset within the array. The

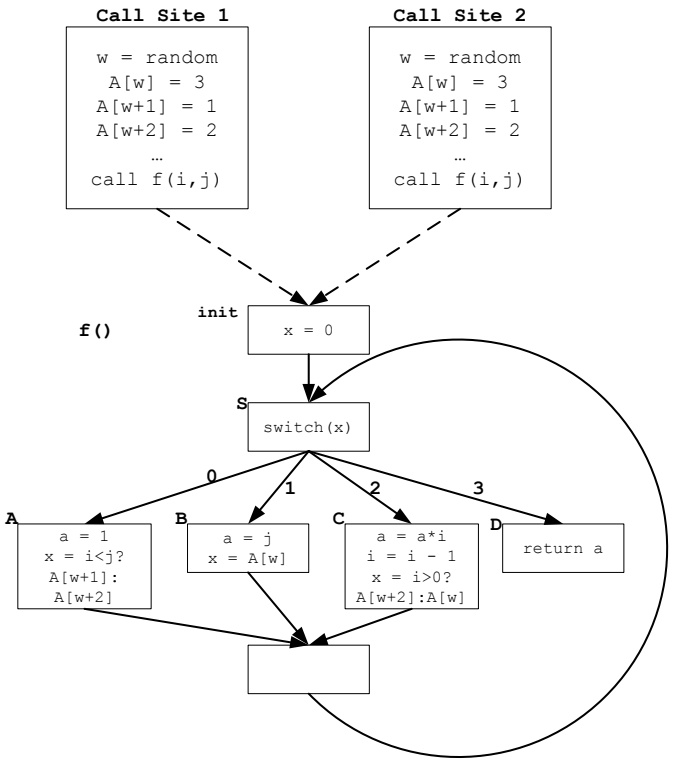

Figure 2.4: After adding inter-procedural data flow (after [125])

offset may be different at different call sites for the function, and is passed to the obfuscated callee either as a global or as an argument. The obfuscated code then assigns values to the dispatch variable from the global array. The code resulting from applying this step to the program in Figure 2.2 is shown in Figure 2.4.

Incorporating dummy blocks and aliasing In order to further confuse the attacker, the previous step can be extended by incorporating dummy basic blocks to the flattened control-flow that never get executed. However, it is still difficult to determine the dummy blocks by static analysis since the indirect branch targets computed by the dispatcher is dynamic. Additionally, indirect load and stores through pointers can be added into the unreachable dummy blocks to make the obfuscated code more difficult to statically analyse. This step is illustrated in Figure 2.5.

Chow et at.'s flattening transformations are somewhat more elaborate. A brief description of their six basic flattening steps follows:

Basic block splitting Each basic block is split into several pieces, where a piece is a

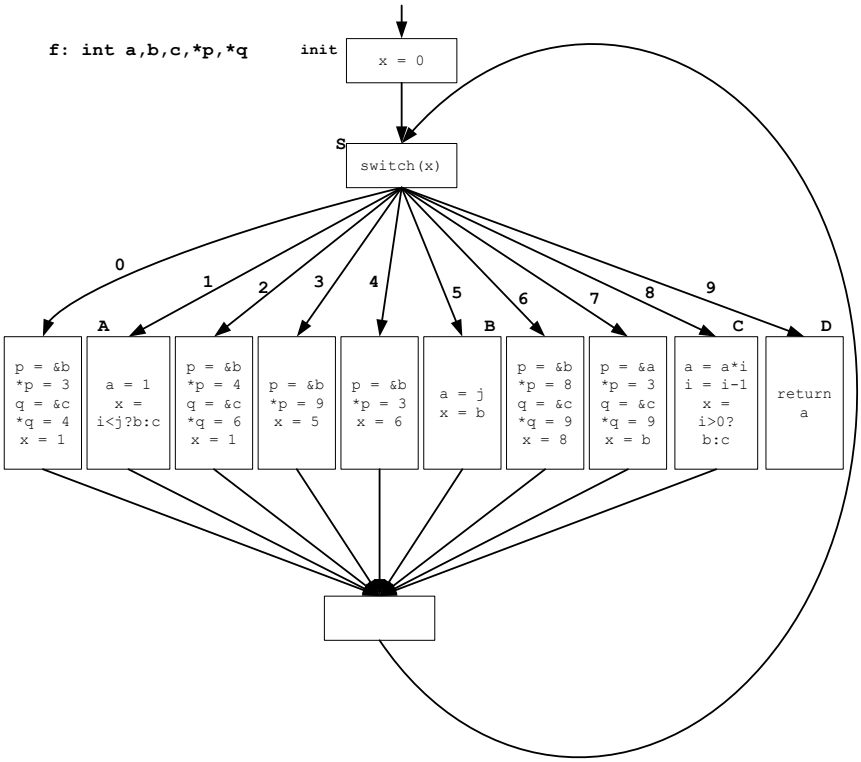

Figure 2.5: After incorporating dummy blocks and aliasing (after [125])

sequence of instructions executed from the first to last.

Introducing dummy pieces Some fake pieces are introduced to confuse the adversary.

Variable renaming Variables in each piece are renamed to names which are unique for that particular piece. Thus, each piece will operated over its own set of renamed variables. A renaming table is generated in this step which keeps the mapping information from old variable names to the new ones.

Connective lump forming A connective lump is generated for every pair of piece generated in the previous step. A connective lump is a sequence of mapping instructions that maintain the association of variables used in one piece with another one in the pair.

Emulative lump forming An emulative lump is formed from several pieces merged together. Each piece may appear in a single lump and all pieces must be used in the lumps. Each time an emulative lump is executed, only a single piece influences upon the computation and the intermediate results computed by other pieces are

discarded. The piece whose intermediate results are retained for further computations is determined dynamically.

Dispatcher lump forming A dispatcher lump is added after all these initial basic block formation steps in order to hide the explicit control transfer between lumps. The dispatcher is added to the beginning of the flattened program. It evaluates some control function and jumps either to the emulative lump whose pieces are to be executed next or to the connective lump to join successive pieces.

In both of these methodologies,, we see that the dispatcher module is the most important component in the flattened (obfuscated) program. In Wang's technique, each flattened basic block while exiting, changes the dispatcher variable through complicated pointer manipulation on some global data structure. Chow *et al.* 's technique, on the other hand, views the dispatcher as a deterministic finite automaton that determines the overall control-flow of the obfuscated flattened program from a state space of given transitions. However, if the state space is rather small, it will not be difficult to deobfuscate the dispatcher. For this reason, its authors expand the state space by incorporating numerous dummy states.

It is worth noting that the theoretical basis rests on a model of a limited adversary. Essentially the idea is to argue that it is infeasible to attack with general-purpose static analysis tools, where the argument proceeds along the following lines [27, 28]: alias analysis tools are necessarily inexact (precision-, efficiency-, scalability-wise), due to the `NP-completeness` of the general problem. Thus a class of obfuscation transforms using pointer aliasing should successfully resist attacks, if these transformations are well engineered to avoid attacks which do not require program analysis, such as pattern-matching.

2.3.2 Opaque predicates

An opaque predicate is a conditional expression whose value is known to the obfuscator, but is difficult for an adversary to deduce statically. A predicate Φ is defined to be *opaque* at a certain program point p if its outcome is only known at obfuscation time.

Following Collberg *et al.* [28], we write $\Phi_p^F(\Phi_p^T)$ if predicate Φ always evaluates to False (True) at program point p for all runs of the same program. We call such predicates *Opaquely True (False)* at program point p. The notation $\Phi_p^?$ is used to denote *Opaquely Unknown* predicate, i.e. one whose value depends on a program input supplied by the user, by the operating system, or by some program such that it sometimes evaluates to True and sometimes to False during different program executions. The opaqueness of such predicates is necessary for the resilience of control-flow transformations.

The following is our proposed taxonomy of opaque predicates based on the properties of the underlying invariants that are used to construct them (our taxonomy is influenced by Collberg *et al.* [27] work).

Algebraic predicates

Algebraic predicates often have invariants that are based on well-known mathematical axioms. A predicate Φ of this class is

$$\Phi : [(x(x+1)\%2 == 0]$$

which is opaquely True for all integers.

To an adversary having previous knowledge about the embedding of algebraic predicates in the program, static analysis attack over the obfuscated code will simply be reduced to code pattern matching. Since it is unlikely that conventional non-scientific software will perform complex algebraic manipulations, a typical attack model would involve searching for control-statements that have algebraic constructs as conditionals and replacing the conditional blocks with straight line code in the True (False) path of the Opaquely True (False) predicates.

Random nondeterministic predicates

Opaquely non-deterministic predicates are based on some function parameter selected at random [131]. The basic method of generating such predicates is demonstrated in Figure 2.6. If x be an integer variable passed as a parameter to a method, the predicate

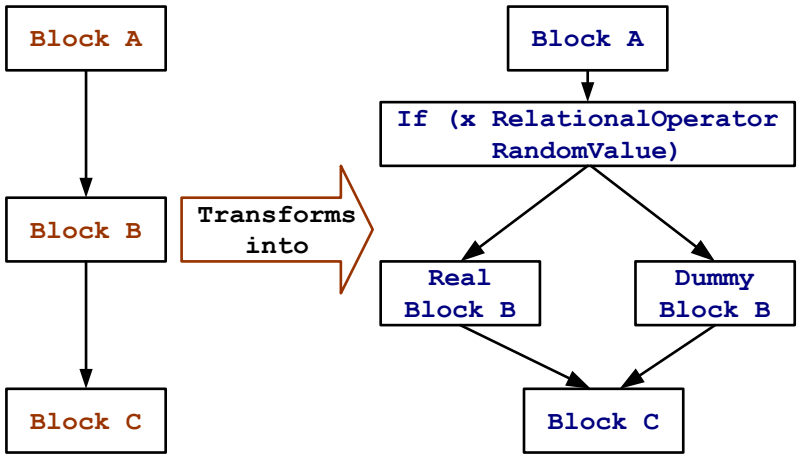

Figure 2.6: Usage of random nondeterministic opaque predicate. Opaque predicates stated as the conditional guards the real block B against a dummy code block (after [131])

$\Phi : [x > RandomInteger]$ corresponds to a predicate of this class. It should be noted that this class of predicates is resilient to trivial pattern matching attacks since the adversary will not gain sufficient information regarding the outcome of these predicates just by recognizing the structure of the predicate. The stealthiness of these predicates will be increased greatly if their random integers are drawn from a probability distribution which resembles the values of integer constants typically observed in programs.

However, if the adversary is able to slice the obfuscated code using x as an input criterion to a static slicer, he/she will be able to trace out the code responsible for maintaining the invariant of this predicate. If the code is statically analysable, the invariant will reveal, among other things, the range of possible values x can be assigned and the adversary will thus be able to statically determine the outcome of $[x > RandomInteger]$ form of random opaque predicates.

Opaque predicates based on aliasing

In Section 2.3.1, we surveyed Wang's [133] technique of constructing dynamic dispatchers using the `NP-hard` pointer aliasing problem for obfuscating control-flow of programs. Collberg *et al.* [27] used this intractability property of pointer aliasing to construct resilient opaque constructs. Their construction is based on the fact that it is impossible for approximate and imprecise static analysers to detect all aliases all of the time.

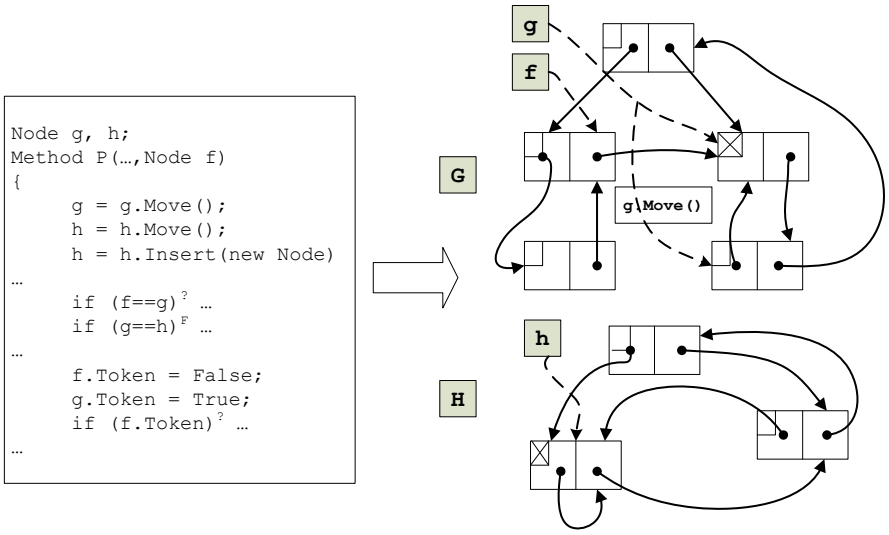

Figure 2.7: Opaque predicates constructed from objects and aliases (after [27])

The basic idea is to construct a dynamic data structure and maintain a set of pointers on this structure. Opaque predicates can then be designed using these pointers and their outcome can be statically determined only if precise inter-procedural alias analysis can be performed on this complicated data structure. Figure 2.7 is an adaptation of Collberg *et al.* 's technique. Node `Node` could be a defined as a doubly-linked list class with fields `String item`, to store its value and two pointers `Node prev` and `Node next` that point to the preceding and succeeding nodes respectively. Method `P`'s control flow is obfuscated using aliased opaque predicates. The method calls manipulate two global pointers `g` and `h` which point into different connected components (`G` and `H`) of a dynamic data structure such as a linked list. The statement `g=g.Move()` will update `g` to move to a different location within `G`. The statement `h=h.Insert(new Node)` inserts a new node into `H` and updates `h` to point to some node within `H`. Method `P` and other methods that call it is also given an extra pointer argument `f` which refers to objects within `G`. Opaque predicates like Φ : **if** $(f == g)^?$ may either be True or False since `f` and `g` move around within the same component. `g==h` must be False since `g` and `h` alias to nodes within different components.

Collberg *et al.* 's definition of the resilience of an opaque predicate does not take into account dynamic analysis attacks. If an attacker can monitor the heap, registers, etc. during execution, then it may be revealed that a given predicate always evaluates to

Predicate	Run 1	Run 2	Run 3	Run 4	Run 5
Φ_1	T	T	F	T	F
Φ_2	T	T	F	T	F
Φ_3	T	T	F	T	F
Φ_4	T	T	F	T	F
Φ_5	T	T	F	T	F

Table 2.1: Palsberg's correlated dynamic opaque predicates

True or False. Palsberg *et al.* [103] attempted to extend the static resilience of Collberg's opaque predicates to withstand dynamic analysis attacks. They observed that in order to protect against such dynamic debugging attack, the obfuscator needs to avoid that a predicate always evaluates to the same result. They proposed using *dynamically* opaque predicates, which are a family of correlated predicates which all evaluate to the same result in any given run, but in different runs they may evaluate to different results. Table 2.1, with five correlated predicates, illustrates their notion of dynamic opaque predicates.

It is an open problem to construct correlated dynamic opaque predicates following Palsberg *et al.* 's description and will be part of our future research. It could have the following structure:

$$\textbf{if } (\Phi_1)\ S_1;$$
$$\textbf{if } (\neg\Phi_2)\ S_1';$$

where S_1 and S_1' are variant versions of the same code block which are difficult to merge (for some reason). We have identified two critical lines of attack.

Pattern match If Φ_1 and Φ_2 are unstealthy that they are recognisable as opaque predicates with correlation 1.0, then the pattern "**if** $(\Phi_1)\ S_1$; **if** $(\neg\Phi_2)\ S_1'$;" may be replaced safely by S_1.

Proving semantic equivalence If analysis can prove $S_1 \approx S_1'$ and $\Phi_1 \Leftrightarrow \Phi_2$ (i.e. these predicates are fully correlated, then this obfuscated structure may be replaced safely by S_1).

Palsberg *et al.* [103] suggested constructing dynamic aliased opaque predicates in watermarked programs from PPCT (Planted Plane Cubic Tree), which is a dynamic data

structure used to store watermark. In doing this, only the part of the program that comes after building the PPCT can be obfuscated using the opaque predicates. A problem here is that this will not obfuscate the code building the PPCT. The authors suggested an improvement by inserting code for building a second random PPCT at the beginning of the original program, and then use that as a basis for obfuscating the code that builds the original PPCT. The authors point out that this results in a chicken-egg conundrum because now the code that builds the random PPCT is unobfuscated and therefore a target for an expert attacker. Such an attacker might locate the code for building the random PPCT and from there unravel the whole construction.

Opaque predicates based on concurrency

Parallel programs are more difficult to analyse statically than their sequential counterparts because of interleaving semantics: n parallel atomic statements can execute in $n!$ ways (and more possibilities if statement executions can overlap). Parallel semantics can be incorporated in an otherwise sequential program using light weight processes called *threads*. If asynchronous events dictate the scheduling policy of threads, a large amount of indeterminacy may be generated and this property could be used introduce opacity. Collberg *et al.* [27] introduces preliminary ideas for constructing opaque predicates using the concurrency of threads. As with aliasing, a global data structure is created and occasionally updated by concurrently executing threads. An example of this could be the data structure created in Figure 2.7. The threads would randomly mode the global pointers `g` and `h` around in their respective graph components by asynchronously executing calls to `move` and `insert`. Their illustration has the advantage of using aliasing with concurrency.

2.4 Conclusion

In this chapter, we started with the existing definitions for obfuscation, namely those proposed by Collberg *et al.* and Barak. We discussed few of the evaluation properties for obfuscating transforms and then provided a brief taxonomy of obfuscating transforms. We

concentrated on the categories of obfuscations which we will use extensively for the rest of the thesis — data and control-flow obfuscations. For data obfuscations we discussed splitting and merging transformations for scalar and composite variables. Following this discussion, we made the observation that control-flow obfuscations in the existing literature have been designed on the intractability results of static alias analysis. We then focussed on a specialised control-altering construct called opaque predicates.

We have noted that obfuscatory strength cannot be guaranteed, in part because it is not known how to arbitrarily generate hard problem instances. Furthermore, the techniques which use hard complexity results as their theoretical basis are "wrong-way" reductions, from a complexity-theoretic perspective. These reductions explain why we should not expect to have exact static analysis tools that will work on all programs, however they do not prove the hardness of deobfuscating any specific output of any specific obfuscation system. Even so, an obfuscation system will have great practical importance if it will resist some known attacks for at least as long as it would take to replace an obfuscated program by a differently-obfuscated program [132].

Following this observation, we focus on evaluating the obfuscatory strength of obfuscations in the next chapter. We will see how available static program analysis tools can be used to deobfuscate programs and in the process of doing our experimentation, we will note that the concept of deobfuscation a program is very context-sensitive in nature. We aim to develop an attack model of obfuscation in the subsequent chapters so that resilient obfuscations can be designed within the assumptions of the attack model to withstand deobfuscation attacks.

3

Evaluation of Obfuscatory Strength

OBFUSCATING transforms discussed in the previous chapter were designed with the ambitious goal of being resilient against "all" possible reverse engineering attacks. While reverse engineering may possibly encompass every program analysis technique, it can be broadly classified into static program analysis and dynamic program analysis. Static analysis is a form of program analysis technique where interesting properties of program code are inferred without actually executing the code. Dynamic program analysis, on the other hand, requires profiled code to be executed and then analysis be performed on trace dumps obtained from the profiled program. Although arbitrarily generating hard problem instances is not known, we still need to be able to evaluate the obfuscatory strength of such transforms in order to corroborate the resilience guaranteed in theory with what actually holds in practice.

The contribution of this chapter is in describing an attack process which an adversary is believed to undertake while reverse engineering an obfuscated program. It also provides a simple empirical evaluation of the strength of obfuscating transforms by considering three different program analysis tools in addition to a human adversary as its attack model. The experience grained from the experimentation will help in choosing a restricted attack model which forms the basis for designing and evaluating obfuscations for the rest of the thesis.

3.1 Background

Before describing our experiments using different static analysis tools, we provide a brief overview of reverse engineering and previous research on comprehending obfuscated code.

3.1.1 Reverse Engineering and deobfuscation

The field of reverse engineering was described formally in the context of software engineering by Chikofsky and Cross in [21]. They noted that the term "reverse engineering" originated in the context of analysis of hardware designs, where deciphering complicated hardware designs from finished products is quite common for improving on the design of finished products or to understand competitor's product in commercial and military context.

Reverse engineering from a software system's point of view is the process of analysing the subject system with the goal of identifying the system's modules and their interrelationships. Forward engineering, on the other hand, can be related to systems development process of moving from high-level abstractions and logical implementation-independent designs to physical implementations of the system. Figure 3.1 shows the interaction between these two. Reverse engineering involves the process of extracting design artifacts or synthesising abstractions that are less dependent on implementation. Chikofsky and Cross observed that the process of reverse engineering does not involve changing the subject system itself or creating a new system based on the subject system — it is merely a

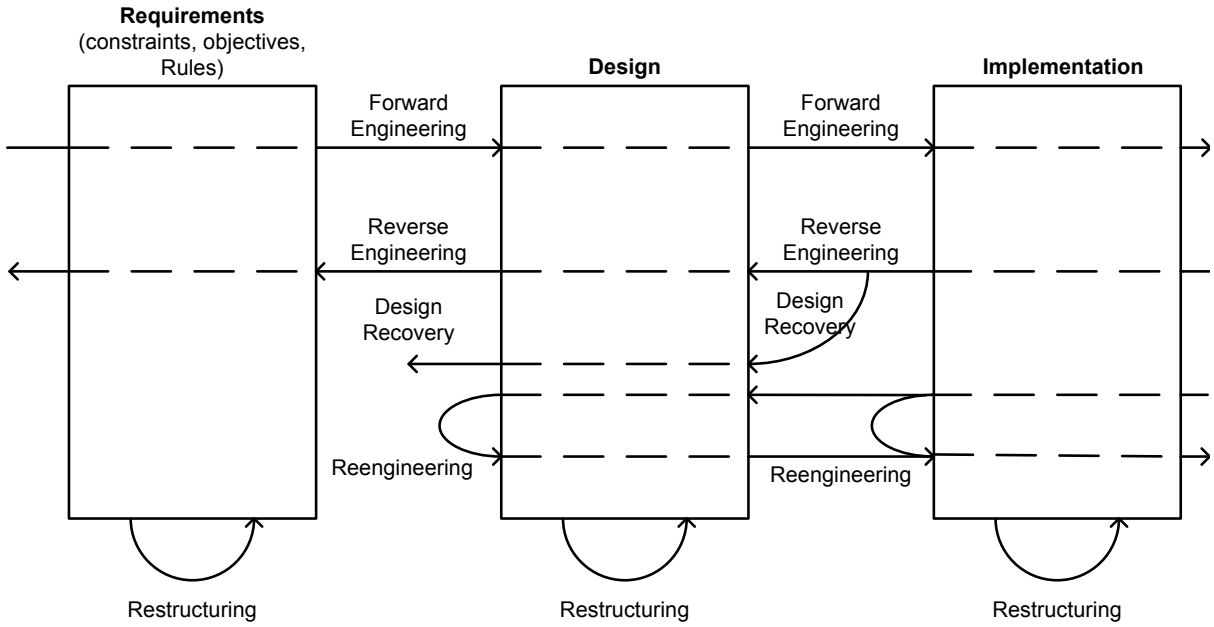

Figure 3.1: Relationship between forward and reverse engineering [21]

process of examination.

But, does deobfuscation also mean reverse engineering? Deobfuscation is a collection of (static and dynamic) analyses for understanding an obfuscated software. The understanding can be said to be successful if deobfuscation leads to a program structure similar to the original unobfuscated program; in other words, the comprehensibility of the deobfuscated program and the original unobfuscated program are same (for some measure of comprehensibility). Indeed, this is also the primary goal of reverse engineering; i.e., to increase the overall comprehensibility of the software system under consideration. The ability to comprehend a software system is necessary in many instances — it could help in coping with complexity of the system itself, generating alternate views of the system, detecting side effects of the system, or facilitating its reuse. We conjure that the process of deobfuscation and reverse engineering are essentially the same, the only difference is in the context in which these two terminologies are used. Deobfuscation is implied in a malicious context whereas reverse engineering in considered as a useful practise in the field of software engineering.

3.1.2 Obfuscatory strength evaluation

The methodologies outlined in Collberg *et al.* [27] gave way to several obfuscation techniques that are based on intractability results of hard computational problems (outlined in the previous chapter). However, we do not know, in practice, how to arbitrarily generate sufficiently hard obfuscated problem instances such that all program analysis techniques would *fail* (e.g. give imprecise, unanalysable results, or be unscalable, run out of memory, crash, or never terminate). So, how do we evaluate obfuscatory strength of such transforms? The main hindrance is in the difficulty of answering the following two questions:

1. *Can general purpose program analysis tools be used to assess the obfuscatory strength of transforms or do we need to develop customised analysis tools instead?*

2. *Can we guarantee that a particular obfuscating transform that can "withstand" attack mounted by a particular tool can also defeat improved versions of the tools (and other general tools belonging to the same categrory) in the future?*

Chenxi Wang in her PhD thesis [132] first made an attempt to empirically evaluate the resilience of her control-flow flattening obfuscations. She used C alias analysis tools NPIC [53] and PAF [48] to analyse programs flattened with her control-flow flattening obfuscating transforms . Wang observed that both tools were successful in analysing small sample programs but failed for large programs in the SPEC benchmarks. She noted in [133] that the analysis failure was inconclusive of whether the tools failed to analyse the artificial aliases introduced in the code or the programs were unscalable for analysis in general. She noted that even for small programs, both NPIC and PAF performed very conservative analysis (any pointer variable is possibly aliased to every variable that appeared in the left hand side of an assignment statement). NPIC was more precise than PAF but failed at resolving aliasing associated with array indices. Udupa *et al.* [125] and Madou *et al.* [79] observed that it is much easier to analyse obfuscating transforms if a combination of hybrid analyses of static and dynamic methods is used. Their motivation of analysis obfuscations was motivated by the problem of analysing obfuscated

polymorphic virus code. They took Wang's control-flow flattening obfuscation as a candidate for evaluation (since it is used commercially) and performed a series of analyses on programs obfuscated with it. Some of these analyses used *cloning* — portions of spurious execution paths were cloned for the purpose of separating them from the original execution paths, *static path feasibility analysis* — a constraint based analysis to find out whether an acyclic execution path is feasible, and *hybrid* — dynamic analysis techniques such as program tracing and edge profiling were used in conjunction with conventional static analysis techniques. They implemented these techniques in tools such as DIABLO and LOCO [80] and reported successful detection and deletion of spurious control-flow edges added by Wang's obfuscation. Lastly, Nakamura *et al.* [94] used human factors experiments to evaluate obfuscatory strength of obfuscations. They used simple layout and loop-fusing obfuscations on two candidate Java programs and reported their degree of comprehensibility by making eight Java programmers simulate the programs mentally (called the Virtual Mental Simulation Model). The experiment was done on source code level and the authors devised metrics in terms of human psychological factors (long term and short term memory) to interpret the results. They observed that certain layout obfuscations reduce the comprehensibility of programs by adding backtracking cost to their mental simulation model.

3.2 The attack process

For the purpose of evaluating the strength of an obfuscating transform, we need to be able to observe the outcome of an "attack" on programs obfuscated with an instance of that transform. Additionally, we also need to draw conclusions about the strength of the transform from the outcome of its attack. The process is outlined in the flow diagram of Figure 3.2.

In the first step of this process, a candidate program is chosen to be obfuscated with the transform which is being tested for its obfuscatory strength. Choosing the correct candidate program is important since not all transforms are suitable for obfuscating every

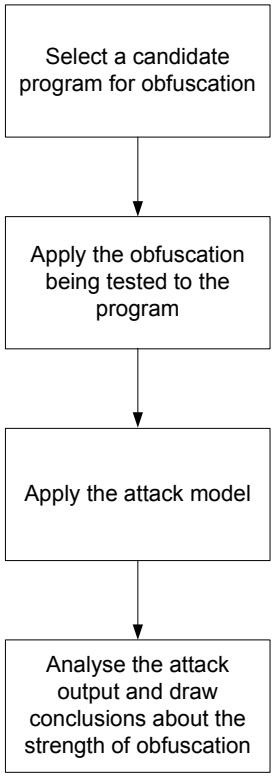

Figure 3.2: An overview of the attack process

program. For instance, a program which is a collection of library calls (such as a system state monitoring program) is not a suitable candidate for data obfuscations for its lack of abundant variables and data structures.

In the second step of the process, the obfuscating transform under study is applied to the candidate program. There are three choices to be made here: the obfuscation can be applied at the source code level, the intermediate code level, or at the machine language level. Obfuscations at the source code level can either be applied manually by hand (if the code is easy enough to parse by a human obfuscator) or by using source code transformers such as TXL [30]. At the intermediate code level, obfuscations can be automatically applied using tools such as DashO [61] and SandMark [96]. Obfuscations can also be applied at the machine code level using binary rewriting frameworks such as DIABLO [98]. This choice is necessarily influenced by the input of the attack model used in the third step of the process.

The third step basically consists of applying an "attack model" to the obfuscated program. Intuitively, a attack model would consist of a human adversary applying program

analysis tool(s) to the obfuscated program. The input to the program analysis tool(s) is the obfuscated program obtained from the previous step and the output is a representation of the input program either in textual or non-textual format which the human adversary would be able to comprehend (if required, taking the help of other tools). This is closely linked with the last step of the attack process where the human adversary derives conclusions about the strength of the obfuscating transform from the output obtained from the previous step of the process. Two issues that need to be considered with respect to an attack model are:

1. Assumptions of the attack model: this includes what the human adversary is allowed (or not allowed) to do with the program analysis tool and the program(s) obfuscated with the transform that is being evaluated. Here, we should also consider if the adversary has access to other information (about obfuscation, original program) or tools that can aid the attack process.

2. Output of the attack model: The output could be a boolean value or a set of metrics. Boolean output is expected for transforms that are expressed in terms of a predicate (such as an opaque predicate) — whether a particular predicate evaluates to true or false value at a program point is expected to give a boolean outcome. On the other hand, an attack tool is expected to give output in the form of metric values if the particular obfuscation being tested, for example, affects a certain scope of the program and the attacker is interested in knowing the coverage of the transform (percentage of code affected by the transform).

In the next section, we take the role of a human adversary and perform this four-step attack process on three programs obfuscated with different transforms. Two issues with respect to the attack model will be discussed for each of the experiments. It is worth noting that the programs we select are simple enough to manually obfuscate at the source code level. The transforms are chosen to reflect two extremes of obfuscation — layout transforms which are considered to be the weakest of obfuscations and aliasing transforms which are considered to the be strongest of obfuscations. The purpose of

our experimentation is to see whether an adversary, armed with a representation of the current state-of-the-art program analysis tools, can evaluate the strength of obfuscating transforms.

3.3 Developing an attack model

As outlined in the previous section, an attack model consists of a human adversary in the loop exploring the use of several program analysis tools to evaluate the strength of obfuscating transforms. For the subsections to follow, we take the role of the human adversary and apply three different program analysis tools on programs obfuscated with two transforms. The program analysis tools are static in nature in the sense that they do not require to execute the obfuscated program in order to perform the analysis.

For each of the subsections, we will describe four steps of the attack process, viz. the selection of the candidate program to obfuscate, the obfuscating transform that is being evaluated for strength, the attack model (what the human adversary can or cannot do with the given program analysis tool), and interpretation of the data obtained from the analysis.

3.3.1 Concept lattices

Tyma's patent [124] and his *Dotfuscator* [62] and *DashO* [61] products, for obfuscating Microsoft Intermediate Language (MSIL) [31] and Java Bytecode [32] respectively, transform identifiers of multiple methods into an identifier of smaller length. The objective of this obfuscating transform is to change the layout of the program and confuse an adversary into thinking that the obfuscated program is semantically different from its unobfuscated counterpart [27]. We take a simple gcd calculating code in Java shown in Figure 3.3 and perform identifier renaming obfuscation following [25]. The resultant obfuscated code is showin in Figure 3.4.

As an attack model, we consider a human adversary armed with the refactoring tool KABA (Klassen Analyse mit Begriffs Analyse — Class analysis with concept analysis)

```
public class test1{
    private int term1;
    private int term2;
    private boolean areRelativelyPrime;

    public test1(int term1, int term2){
        this.term1=term1;
        this.term2=term2;
        areRelativelyPrime=areRelativelyPrime();
    }

    public static int gcd(int term1, int term2){
        int remainder;
        remainder=term1%term2;
        if (remainder==0){
            return term2;
        }
        else{
            return gcd(term2, remainder);
        }
    }

    private boolean areRelativelyPrime(){
        if (gcd(term1, term2)==1){
            return true;
        }
        else{
            return false;
        }
    }

    public static void main(String args[]) {
      test1 a=new test1(12, 19);
   }
}
```

Figure 3.3: A simple gcd calculating test code.

[119]. KABA uses the Snelting-Tip concept analysis algorithm [115] in order to determine a behaviour preserving refactoring transform which is optimal with respect to a given set of instance usage. KABA is an interesting attack tool for this particular obfuscating transform since it produces output in the form of concept lattices. Snelting and Tip [115] proposed the use of concept lattices in software engineering domain for understanding class hierarchies in object-oriented languages. Thus, using concept lattices produced by KABA, we are interested to see whether identifier renaming obfuscation induces any structural change (with respect to instance usage) in the obfuscated code.

```
public class a{
    private int a;
    private int b;
    private boolean c;

    public a(int a, int b){
        this.a=a;
        this.b=b;
        c=c();
    }

    public static int b(int a, int b){
        int c;
        c=a%b;
        if (c==0){
            return b;
        }
        else{
            return b(b, c);
        }
    }

    private boolean c(){
        if (b(a, b)==1){
            return true;
        }
        else{
            return false;
        }
    }

    public static void main(String args[]) {
      a b=new a(12, 19);
    }
}
```

Figure 3.4: Obfuscated code using identifier renaming.

The human adversary in this particular experiment is assumed to only use KABA for analysis of programs in Figure 3.3 and 3.4. Note that here the assumption is relaxed from the usual situation where the adversary has access to only the obfuscated program and not the unobfuscated one. However, since we are interested in evaluating layout obfuscation and in knowing whether the instance access patterns in the obfuscated program is different from the access patterns of the original program, we need the original program as a basis for comparison.

An overview of the Snelting-Tip concept analysis algorithm is provided in the following

four phases. Details can be obtained from [115].

Collection of member accesses In this first step, all field and method accesses in the given source hierarchy and its client set are collected and set up as a table. The rows are labeled with variable names and the columns are labeled with fields and methods from the hierarchy. An access $o.m()$ from an object leads to a table entry for $(o, C.m)$ (where C is the static class for m and o is an object reference). For methods, there is a distinction between declarations and definitions (i.e. implementations). Points-to analysis is used to determine for an object reference o to which object creation sites it might point to at runtime. The static analysis of KABA can handle full Java bytecode. Intraprocedural points-to analysis is flow-sensitive and can be parameterised to be context- and object-sensitive.

Incorporation of type constraints In order to guarantee preservation of behaviour, a set of type constraints is extracted from the source and incorporated into the table from the first phase. More entries are then added, until a minimal table is obtained which respects all constraints.

Generation of concept lattice Concept analysis generates a lattice from the table obtained from the second phase. The lattice represents the same information as the table, but is organised into a hierarchical view. Every lattice element represents a class, and common fields or methods are factored out into super-classes. It helps in providing a fine-grained insight into member access patterns for all objects and variables.

Simplification of concept lattice In the last phase, the concept lattice size is compressed by a considerable amount so that a manual inspection of it is comprehensible. The user may contribute background knowledge during lattice simplification in order to control the structure of the final refactoring.

We illustrate Snelting-Tip algorithm with the simple example of Figure 3.5 adapted from [119]. Class B is a subclass of A, redefines $f()$ and accesses the inherited fields x, y.

```
public class A {

    int x, y, z;
        void f() {
            y = x;
        }
    }

    class B extends A {
        void f() {
            y++;
        }
        void g() {
            x++;
            f();
        }
        void h() {
            f();
            x--;
        }
    }

    class Client {
        public static void main(String[] args) {
            A a1 = new A(); // A1
            A a2 = new A(); // A2
            B b1 = new B(); // B1
            B b2 = new B(); // B2

            a1.x = 17;
            a2.x = 42;
            if (...) { a2 = b2; }
            a2.f();
            b1.g();
            b2.h();
        }
    }
}
```

Figure 3.5: A example program

The main program creates two objects of type A and two objects of type B, and performs some field accesses and method calls.

Figure 3.6 presents the simplified concept lattice which also forms a refactoring proposal. Lattice elements are the classes of the new hierarchy. They are marked with class members above, and with variables or objects below. The members above an element (i.e. a new class) define the new class' members; variables below an element will obtain this element as their new type. The concept lattice illustrates different member access pat-

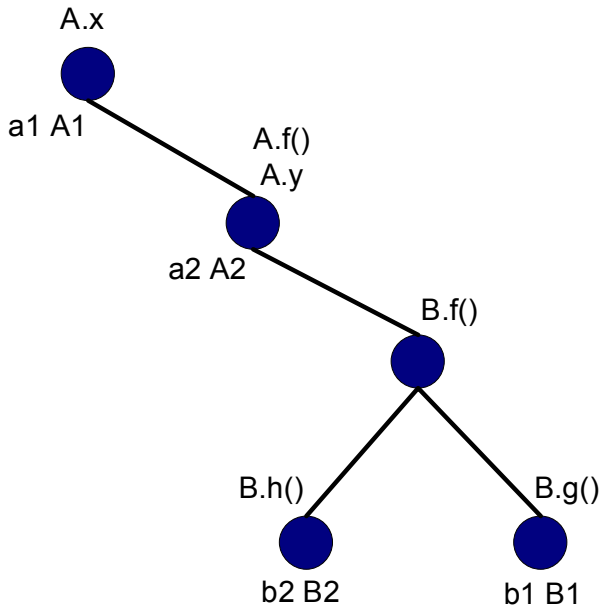

Figure 3.6: Concept lattice for program in Figure 3.5.

terns of the objects in the program. Typically, a class is split into new unrelated classes if there are objects which access one subset of its members, and other objects which access another, disjoint subset of its members. New subclasses and inheritance relations are introduced if there are objects accessing only a subset of a member set accessed by other objects. The following observations from [119] can be made from the concept lattice of Figure 3.6:

- The two objects of class B have different behaviour, as one calls g and the other calls h. Therefore, the original class B class is split into two unrelated classes.
- The two objects of original type A have related behaviour, as $A2$ accesses everything accessed by $A1$ and also $A.f()$. Therefore, the original class A is split into a class and a subclass.
- $A1$ only contains $A.x$ but not $A.y$. $A.z$ is not live and does not appear in the concept lattice.

Figure 3.7 shows the KABA concept editor display in the form of a concept lattice for the program in Figure 3.5. Every box represents a class, its name is printed in bold font in the centre (nodes containing only members from the same original class C are named

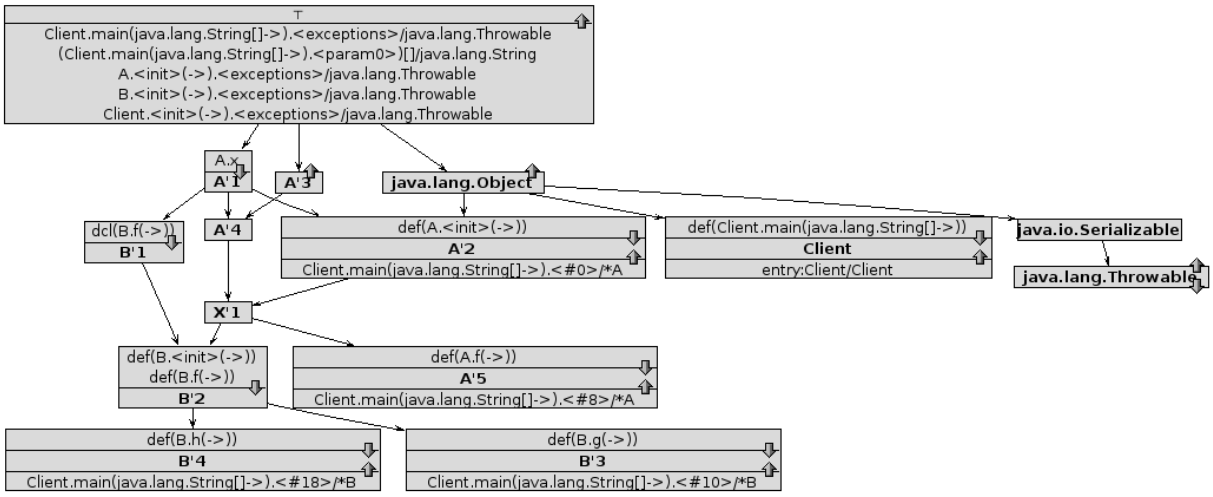

Figure 3.7: Concept lattice of program in Figure 3.5

$C'n$; users have the option to manually rename classes). Members are displayed above the class name, variables below it. To reduce the screen space requirements, attributes and objects are not displayed by default; scroll arrows next to the class name allow the to expand them if necessary.

Figure 3.8 shows the corresponding concept lattice for the code in Figure 3.3. `term1` and `term2` have been factored out in `test1'1` and `areRelativelyPrime` in `test1'2`. Because of the use of fields `term1` and `term2` in `test1()` and field `areRelativelyPrime` in `areRelativelyPrime()`, both methods have been defined in a lower hierarchical order at `test1'3`. Figure 3.9 shows the concept lattice for the obfuscated code in Figure 3.4. It has the same hierarchical structure and factors out variables and methods in the same manner as for code in Figure 3.9.

Referring back to our attack process, the output obtained from KABA shows both the candidate program and its obfuscated counterpart are structurally similar with respect to their concept lattices, reflecting that identifier renaming fails to change the way instances are used in the original program and its obfuscated counterpart. Considering that the objective of obfuscation is to make the obfuscated code *unintelligible* (i.e. the obfuscated code reveals less information than its unobfuscated counterpart when analysed under the same attack model), we can say that identifier renaming obfuscation fails against an attack tool such as KABA. The obfuscated code might "look" obscure with because of renaming

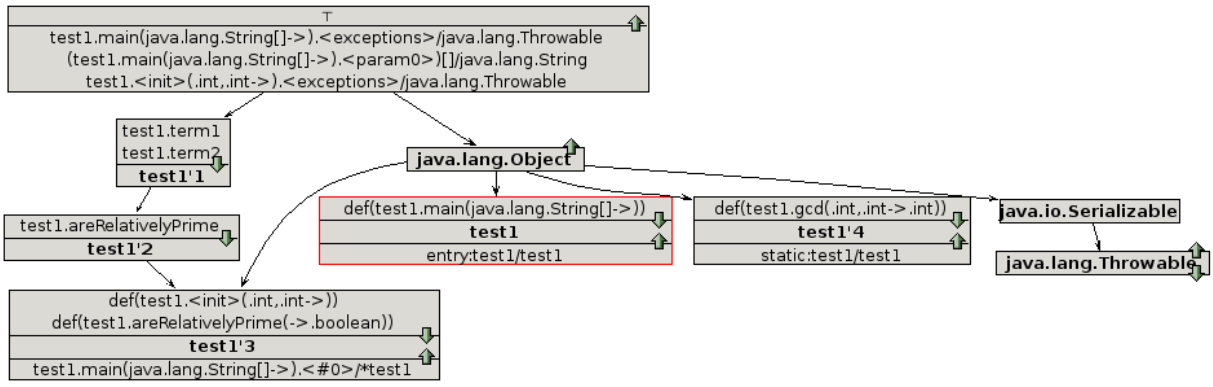

Figure 3.8: Concept lattice of program in Figure 3.3

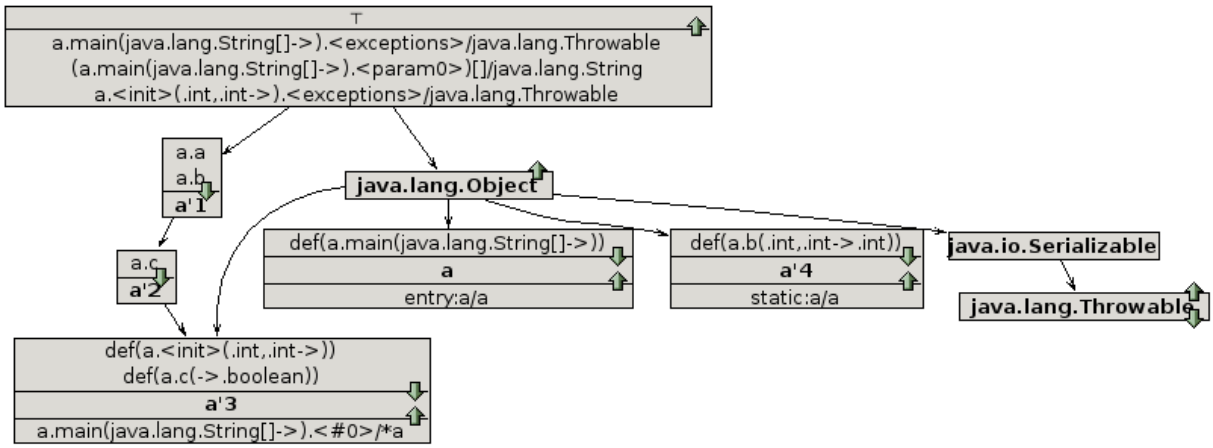

Figure 3.9: Concept lattice of obfuscated code in Figure 3.4

identifiers but it is equally analysable as the unobfuscated code in the sense that same amount of information is available from both the concept lattices.

3.3.2 Points-to analysis

In the previous chapter we identified *opaque predicate* as a conditional expression whose value is known to the obfuscator, but is difficult for an adversary to deduce statically. Collberg *et al.* illustrated an interesting control-flow obfuscation in [28] where intractability results of pointer aliasing were used as a basis for designing opaque predicates (see 2.3.2). In this section we attempt to evaluate the strength of pointer aliasing that form the basis of such aliased opaque predicates. Before describing the experiment, we provide a brief overview of points-to analysis for Java.

Background

Points-to analysis computes for every pointer the set of objects it may point to at runtime [41, 54] . For Java, there are no pointer arithmetic or no pointers to pointers, and type casts are type safe. Therefore, points-to analysis for Java is quite different than for the C language [105] . In this section we will provide the basics of points-to analysis algorithms from [120] and then describe an appropriate framework for points-to analysis in Java. Such frameworks are used as "backend" engines for program analysis tools and their efficiency, precision, and scalability directly influence the quality of output of the "frontend" tool.

Let *ptr* denote the set of all pointers (object references in Java); *obj*, the set of all objects (constructor call sites); *assign*, the set of all assignments; $pt(p)$, the points-to set. An assignment of the form `l=r` is written as $(l, r) \in assign$. When a pointer is assigned to an object reference, it leads to the inclusion of the object in the points-to set:

$$(p, o) \in assign \wedge o \in obj \Rightarrow o \in pt(p)$$

Traditional algorithms as proposed by Andersen [2] and Steensgaard [118] treat pointer assignments differently. Andersen uses a subset relationship:

$$(p, q) \in assign \wedge q \in ptr \Rightarrow (pt(q) \subseteq pt(p))$$

While Steensgaard merges the two points-to set:

$$(p, q) \in assign \wedge q \in ptr \Rightarrow ((pt(q) = pt(p))$$

Steensgaard's algorithm is considered to be faster than Andersen's algorithm whereas Andersen's algorithm is considered to be more precise than Steensgaard's one. In order to gain more precision for Steensgaard's algorithm at the cost of some execution time, type constraints can be added to the analysis. For Java, type-incorrect arcs can be eliminated from the points-to set using the following rule:

$$(p, q) \in assign \Rightarrow ((o \in pt(q) \Rightarrow o \in pt(p)) \wedge (o \in pt(p), type(o) \leq type(q) \Rightarrow o \in pt(q)))$$

Interprocedural analysis uses these rules in order to handle assignments to formal parameters, return values, and this-pointers. The analysis of method calls in Java uses the already computed points-to set for the call's target object reference. For any possible target object in this set, the corresponding target method is determined according to the target object's static type. The call to the target method is then treated by taking into account assignments to formal parameters, return values, and this-pointers. Constraints for the resolution of dynamic binding have the following general form:

$$(o \in pt(p) \wedge lookup(o, f) = C) \Rightarrow (..., ...) \in assign$$

where f is the method; *lookup* determines the class which contains the appropriate definition of f according to the type of o.

BDDBDDB as an attack model

BDDBDDB is an implementation of Datalog, a declarative programming language for specifying program analysis. The input to it is Java bytecode of the program which is then parsed by the flow analysis framework of Joeq [128]. The output of Joeq is then queried using Datalog rules and the output of this step produces the points-to set for queried variables. When using BDDBDDB in the attack model, the human adversary is assumed to have access to the bytecode of the program obfuscated with the aliased transform. The adversary can use Datalog to query the Joeq output and derive conclusions about the points-to set given as output for each input variable of the transformed program.

BDDBDDB is *context-sensitive*, which means it can distinguish between different calling contexts of a method and thus prevents erroneous propagation of alias information from one caller to another of the same method. It is also very scalable and can scale up to analysis of nearly 700,000 bytecodes. The analysis is *field sensitive* meaning that it can track individual fields of individual pointers. BDDBDDB is *flow sensitive* meaning it takes into account control-flow information of the analysed program. Context-sensitivity in BDDBDDB is achieved through the use of *cloning*. Cloning generates multiple instances of a method such that every distinct calling context invokes a different instance,

thus preventing information from one context to flow to another. Thus, through cloning, context-sensitivity can be generated using context-insensitive algorithm by applying it to each of the clones. BDDBDDB, however, is not *allocation site sensitive*. Objects are referred by their allocation sites and since allocation sites are not context-sensitive, BDDBDDB approximates one allocation site for every calling context of an object.

In the Figure 3.10, we construct a simple program containing an aliased predicate. With BDDBDDB as our attack model (additionally, a human interpreter of the output of BDDBDDB), we choose to evaluate the outcome of the predicate. BDDBDDB fails to report that the variables a and b alias in the predicate of the example in Figure 3.10. This observation indicates that it might be easy to confuse a BDDBDDB-armed adversary trying to attack aliased opaque predicates that exploit this weakness in BDDBDDB. The drawback, however, of relying on weaknesses of a particular release of a tool is that no guarantee on obfuscatory strength can be made against attack models that use future improved releases of the tool.

Nagra in [93] however commented that determining whether an aliased predicate is opaquely true or false reduces to solving the "must-alias" problem (two variables must alias at a given program point if they always have the same value when control reaches that point [64]). Tools like BDDBDDB are conservative in their approach since they report only "may-aliases" (two variables may alias at a given program point if they have the same value when control reaches that point [64]). When analysing aliased opaquely true (false) predicates, an attacker would be interested in learning if variables constituting that predicate always (do not) alias each other at that particular program point. For answering this question, a tool that performs must-alias analysis is required since a conservative output from a may-alias analysis will not conclusively tell if a predicate is opaquely true or false for all program runs at a particular program point.

```
public class Test {

   public static void main(String[] args) {
      Test x = new Test();

      Test a = (Test) id(x);
      Test b = (Test) id(x);
      if(a==b)
         System.out.println("false predicate");
   }

   static Object id(Object o) {
     Test p = new Test();
     return p;
   }
}
```

Figure 3.10: Example of an opaque predicate needing allocation site sensitive alias analysis tool to correctly determine its value

3.3.3 Program slicing

Even if an attacker fails to determine if an aliased opaque predicate holds true at a certain program point, he/she may still gain significant understanding of predicate variable updates if he/she manages to find out the invariants involved in maintaining the predicate. This attack can be mounted using a program slicer which when given a certain slicing criterion (a statement/variable pair which could be the opaque predicate variable) will slice (by marking) relevant parts of the program that updates the predicate [123]. In this part of the experimentation, we use a slicer in the attack model and try to find out parts of the program that are responsible for maintaining the state of the variables in the obfuscating transform at a certain program point.

We selected the Indus [126] Java static slicer to simulate this attack. Indus was chosen because it uses Soot [130] to derive points-to results [74] and has reasonable analysis precision [65]. Indus incorporates a program slicing library which facilitates different high level program analyses such as dependencies on intra- and inter-procedural data, control, interference, and synchronization of the program. Assumption of the attack model is that the human adversary in possession of both the obfuscated program and Indus can slice the program by providing a slicing criterion. The slicing criterion for Indus could be a source

code line or a Jimple (an intermediate representation used in the Soot [127] framework) statement. The output of Indus is a slice with respect to the slicing criterion (where a slice is defined as a subprogram which computes same value of the variable at the same program point defined by the slicing criterion).

We used the program fragment of Figure 3.11 to simulate a slicing attack on opaque predicates using Indus. Here, we used a predicate statement as slicing criteria. A precise slice of all statements affecting the slicing criteria constitutes a successful attack. In our example we inserted an aliased opaque predicate `if(g != h.selectNode(2))` in a fragment from the SciMark benchmark for LU decomposition, a method for decomposing a matrix into a product of lower triangular and upper triangular matrices. A particular method `solve` has been shown which performs solving by substitution. We inserted updates to the dynamic `Node` data structure in different methods of LU and inserted predicates in `solve`. The methods which update the `Node` structure has been omitted in the illustration. Indus slices the statements (shown in reverseface) `g.addNode(2)` and `h.addNode(1)` in the method `solve`. A simple slicing attack could be mounted with the attacker slicing a program based on an opaque predicate as the slicing criterion. If the slice is input invariant [1], the attacker can safely remove the statements in the slice and replace the predicate with a constant without compromising the correctness of the program.

Collberg and al. [28] proposed a solution for increasing the stealth of opaque predicates by merging classes used in building the obfuscating data structure with the other similar user-defined class. We argue here that this modification will still not resist slicing attacks since slicing is done at the statement level and is not affected by class hierarchy of the program. A solution to increase the stealth of opaque predicate by interweaving the code maintaining the predicate with the rest of the program code is illustrated in Figure 3.12. The predicate maintaining data structure code is correlated with original program data structure by introducing a bogus conditional, which cannot affect our opaque predicate, such as `if(b[1] > b[2]) h.addnode(1)`. Using the same slicing criteria as in Figure 3.11, we get a much bigger slice for the program in Figure 3.12. Thus, by introducing such simple

[1] An input invariant slice does not depend on keyboard input or file read

```
public static void solve (double LU[][], int pvt[], double b[]) {
    int M = LU.length;
    int N = LU[0].length;
    int ii = 0;

  g.addNode(2);

    for (int i=0; i<M; i++)
    {
        int ip = pvt[i];
        double sum = b[ip];

        b[ip] = b[i];
        if (ii==0) {
            for (int j=ii; j<i; j++)
                sum -= LU[i][j] * b[j];
        { else {
            if (sum == 0.0)
                ii = i;
        }
        b[i] = sum;
    }

  h.addNode(1);

    for (int i=N-1; i>=0; i--)
    {
        double sum = b[i];
        for (int j=i+1; j<N; j++)
            sum -= LU[i][j] * b[j];
        b[i] = sum / LU[i][i];
    }

  if (g != h.selectNode(2))
        b[1] = b[1] + 1;
}
```

Figure 3.11: Example of a program slice using Indus. The slicing criterion selected here is the opaque predicate if(g != h.selectNode(2)). Statements in the slice are labeled with reverseface.

correlations, we can make it difficult for the adversary to blindly strip off the sliced code from obfuscated program.

3.3.4 Discussion

After describing the attack process to evaluate obfuscatory strength using three general purpose static analysis tools, we revisit the two questions posed in Section 3.1.2. The first question was related to answering whether general purpose program analysis tools

```
public static void solve (double LU[][], int pvt[], double b[]) {
   int M = LU.length;
   int N = LU[0].length;
   int ii = 0;
   g.addNode(2);

     for (int i=0; i<M; i++)
     {
          int ip = pvt[i];
          double sum = b[ip];
          b[ip] = b[i];
          if (ii==0) {
               for (int j=ii; j<i; j++)
                    sum -= LU[i][j] * b[j]; {
          else {
               if (sum == 0.0)
                    ii = i;
          }
          b[i] = sum;
     }
     if (b[1] > b[2])
          h.addNode(1);
     for (int i=N-1; i>=0; i--)
     {
          double sum = b[i];
          for (int j=i+1; j<N; j++)
              sum -= LU[i][j] * b[j];
          b[i] = sum / LU[i][i];
     }
    if (g != h.selectNode(2))
          b[1] = b[1] + 1;
}
```

Figure 3.12: Example of program slice of the same method used in Figure 3.11 after inserting our correlation code if(b[1] $>$ b[2]) h.addnode(1); . The slicing criteria is same as before. Statements in the slice are labeled with reverseface.

would suffice to reverse engineer obfuscating transforms. From our simple experiments, we can observe that indeed general purpose tools can be used in the attack model by an adversary. Concept lattices produced by KABA showed that a program obfuscated with

layout transformations has essentially the same instance usage structure as its unobfuscated counterpart. A static slicer in the form of Indus showed that it is relatively easy for an attacker to find out the portions of the code responsible for maintaining the state of the obfuscating transform (in our case an aliased opaque predicate). For our second question we observe that even if an adversary is limited by the accuracy and lack of features implemented in current generation points-to analysis frameworks such as BDDBDDB, he/she may be able to attack program obfuscated with same transforms using future improved versions of the tool.

The only interesting conclusion we can draw from our experiments is that it is possible to defeat the human adversary involved in interpreting the output of a static slicer by intertwining the code responsible for maintaining the state of the variables in the obfuscating transform with the rest of the program. Note that this is not a defeat of the slicing algorithm rather an attempt to deter the comprehensibility of attack output by a human adversary.

We realise that the problem of providing a provable security model for obfuscation has plagued the cryptography research community for a long time. The security requirements for obfuscation have been poorly understood and this has resulted in the lack of credible theoretical results in obfuscation since Barak's landmark paper [7] in 2001. Keeping this in mind, we take the complementary approach of addressing the problem by coming up with weaker notions of obfuscation. Whereas the theoretical approach to defining provable security of obfuscations has been to consider security against "all" polynomial-time adversaries, we consider it prudent to consider weaker classes of adversaries, and in the thesis we take the simplest case where the attack model is a human adversary equipped with just a single (a-priori known) static slicer. In order to define success and failure of a particular obfuscation with respect to a slicer, we need some kind of "understandability" metrics. These would be subjective metrics whose values are established by the person who is summarising the results of an experiment, where the understandability is assessed with respect to the person's deobfuscation goal. The understandability metrics need to map obfuscated programs onto the unit interval, i.e. they should be capable of defining

a degree of obfuscation rather and not just say if a transform can be cracked (as with cryptography).

With these two goals in mind, we will design and evaluate obfuscations within the scope of an attack model where a static slicer is solely used as an attack tool. It is worth noting that Ivanov and Zakharov first proposed the idea of designing obfuscations with the intent of obstructing static analysis attacks on code [63]. In their framework of study, a program static analysis tool A computes a mapping of the form: $A : U \mapsto W$, where U is the set of unobfuscated programs under consideration and W is the set of possible outcomes of static analysis. The set W of all possible outcomes forms a lattice which has the minimal element $\perp$. This element can be interpreted as a lack of any useful/significant information about the program semantics as may be analyses using A. They defined an obfuscation $\mathcal{O}$ to be *impervious* to a program static analyser A if $A(\mathcal{O}) = \perp$ holds for any program P. With this goal in mind, a particular static analysis tool can be made *unsuitable* for deobfuscation if the program P is obfuscated using $\mathcal{O}$. They defined the efficiency of a slicing algorithm A_S by the ratio

$$\frac{\#(S^{A_S}_{<p,V>})}{\#(S^{min}_{<p,V>})}$$

where $(S^{A_S}_{<p,V>})$ is the set of program points in the slice with respect to the criterion $< p, V >$, and $(S^{min}_{<p,V>})$ is the minimal set of program points that are necessary to be maintained in order to correctly compute the values of variables V at program point p. The authors noted that an obfuscation could be made impervious to a slicing algorithm if it is forced to give the worst possible slice, i.e., the set of *all* program points that precede p in P.

There are a couple of problems with the imperviousness property. The first one concerns its absoluteness. While it may seem trivial to incorporate dependencies to link up every program point to the slicing criterion, our practical experience would suggest otherwise. An all-or-nothing model lends a cryptographic flavour to the obfuscation problem whereby an adversary managing to obtain a slice with one program point less than the

worse possible slice would be considered a winner. Also, a bogus obfuscation will add spurious amount of dummy code which could then be made to depend on the slicing criterion — this is not really a correct measure of the success of obfuscation. How do we relax the imperviousness goal of obfuscation and how do we design obfuscations that link up the statements that are not included in the slice with respect to a particular slicing criterion? The next two chapters will address these issues. Secondly, Ivanov and Zakharov's technique is not evaluated empirically. It is easier to introduce false obfuscation dependencies in pseudo-code and claim it as being impervious to *any* slicing algorithm. However, in practice, engineering dependencies within program code and controlling side effects of analyses can be quite challenging and therefore it is important to validate how the theoretical claims corroborate when reduced to practice.

3.4 Conclusion

Since Barak highlighted the need for provable security for obfuscation [5], we have witnessed a spate of obfuscation techniques based on the hard complexity problems of program analysis. However, no method of generating hard problem instances is currently known, except for numeric problems such as integer factorization.

A contribution of this chapter is in describing an attack process an adversary would undertake to evaluate an obfuscating transform. The second contribution rests in demonstrating that general purpose program analysis tools could be successfully used in the attack process by an adversary. We also noted that obfuscatory strength of a transform cannot be guaranteed based on the weaknesses of the current implementation of a tool. The third contribution of this chapter is in laying a basis for an attack model to be used for the rest of the thesis. We observed that rather than trying to "defeat" a tool, it would be prudent to restrict the attack model by attempting to defeat the human adversary analysing the output of this tool. For the rest of the thesis, we will consider a human adversary using a static slicer as our attack model.

We noted towards the end of the chapter that we need to be able to come up with a

restricted attack model for obfuscation within which we can define metrics for success or failure of obfuscation. We observed that previously coined metrics, such as the "imperviousness", are absolute in nature and are thus unsuitable for expressing the "degree" of obfuscatory strength achieved by a particular transform. In the next chapter, we will develop a weaker notion of obfuscation with respect to the attack model and design metrics for measuring the degree of obfuscation.

4

Design and Evaluation of Slicing Obfuscations

REVERSE engineering tools and techniques are being used extensively by software pirates to "crack" obfuscated code. Program analysis tools are a subset of reverse engineering tools and even though their use is typically associated with code maintenance operations in software engineering, we demonstrated in the last chapter how different program analysis tools can be used in an attack model to reverse engineer obfuscated. Of the three broad categories of tools used in the previous chapter, we noted that it is a promising step toward manufacturing obfuscated programs if it is possible to confuse a human adversary interpreting the output from a static slicer. We also identified the need to develop metrics to measure the potency of obfuscations and

conduct experiments to see how these metrics vary when obfuscated code is subjected to slicer attacks.

The contribution of this chapter is to discuss slicing attacks in detail and give definition of "slicing obfuscation" — which are transforms for deterring slicing attacks. By considering the maxim: "Attack is the best form of defence", we take the role of an obfuscator who uses obfuscation to try to limit the usefulness of program slicing for program comprehension. Note that the novelty of slicing obfuscations rests in the fact that they are manufactured from inherent program dependencies with a particular attack tool in mind (the reader should not be mislead into thinking that these are "new" obfuscations — the emphasis is not on proposing different obfuscations but on the process of "designing" them from program artefacts). The second important contribution is in development of metrics, based on those from the field of program slicing, for measuring obfuscatory strength of slicing obfuscations. The third contribution of this chapter is in the presentation of empirical data looking at the effect of various obfuscations have on the metrics for a suite of small programs.

4.1 Background

Program comprehension techniques form the basis of reverse engineering since the primary goal of such techniques is to identify *relevant/interesting* parts of the code and create representations of the program at a higher level of abstraction. Indeed, this is what an adversary intends to do when he/she attempts to steal or change some relevant part (of her/his interest) of the code with the intention of reusing it in illegal derivative software or invalidating the code licensing routine. It would seem obvious that a natural way to deter such *code comprehension attacks* is to intertwine the relevant code routine with other irrelevant sections so that the attacker fails to recognise the portions of interest by analysing the code. Before proceeding to describe techniques for intertwining code, we provide background overview of program slicing and its use in program comprehension.

4.1.1 Program Slicing

Program slicing as a software engineering technique was first introduced by Mark Weiser [134] in 1980. Since then, researchers have tried to improve on Weiser's precision and extensibility. Moreover, work has also been done to add functionality to basic slicing algorithms. We discuss in this subsection, a few of the features available in different slicing tools and algorithms. Details can be found in extensive surveys and evaluation reports [123, 55, 13, 137, 10].

A *program slice* consists of the parts of a program that potentially affect the values computed at some point of interest (called a *slicing criterion*) [123]. According to Weiser's original definition, a slicing criterion C consists of a pair (line-number, variable) and parts of a program which have a direct or indirect (computed through data and control-flow analyses) influence on the values computed at C are called the *program slice* with respect to C. The notion of slicing comes from Weiser's observation that abstracting away parts of program corresponds to the mental abstractions that people make when they are debugging a program. In his original proposal, Weiser defined a program slice S as an executable subset of a program obtained from deleting statements such that S replicates part of the behaviour of the program. In subsequent research [60], the executability feature was waived and slices were just defined to be subset of statements and conditionals of the program which directly or indirectly affect the value computed at the criterion C. Weiser's slices were also *backwards*, which means the statements in S are calculated by a backward traversal of the program starting from C. This gives the statements which affect the value of variables in C right before the statement defining C is executed. The notion of *forward* slicing was first defined by Bergeretti *et al.* [9]. A forward slice gives the statements and conditional predicates that depend on the values of variables defined at the slicing criterion C. Tip in [123] observed that both forward and backward slices are computed in a similar way. Unless otherwise stated, slicing in our context would mean backward slicing.

Slicing techniques can be categorised based on their capability to slice statically or

dynamically. Weiser's slicing technique mentioned in the previous paragraph is a static one since the underlying slicing algorithm finds a program statement subset by solving (approximately) a set of static program analyses problems but the program that is being sliced is not executed. The concept of dynamic slicing was proposed by Korel and Laski in [67] and is considered to be an instance of flowback analysis [123]. In flowback analysis, the user interactively traverses the graph that represents data as well as control-flow information and traces the information path that leads to a particular value of a variable in the program. Unlike flowback analysis, however, dynamic program slicing is non-interactive and only the dependencies that occur in a particular run (execution) of a program are taken into account. A dynamic slicing criterion is a triple (input, occurrence of a statement, variable). Dynamic slicing assumes a fixed input for the program, whereas static slicing does not make assumptions regarding the input. A dynamic slicing algorithm, therefore, is thought to give a more precise and smaller slice compared to its static counterpart. Again in our experiments, unless otherwise stated, slicing would mean static slicing. Variations of static and dynamic slicing (such as amorphous and quasi-static slicing) can be found in the survey articles.

Static slicing techniques can be categorised based on their intermediate representation of dependency information. Weiser's algorithm is most primitive and it uses dataflow equations to represent data and control-flow dependencies. The slice is obtained by an iterative solution of the dataflow equations. The slicing criterion in Weiser's case maps to a particular node in the Control-Flow Graph (CFG) of the program. A slice is "statement-minimal" if no other slice exists with fewer statements for the same criterion and it has been shown to be undecidable [134]. Ottenstein and Ottenstein, in [101], first proposed slicing as a graph reachability problem on Program Dependency Graphs (PDG) [42]. A PDG has program statements as nodes and either control or data dependencies between statements as edges. Data dependence edges are labelled with the variable on which the dependence holds, while control dependence edges are labelled with the truth value which determines the execution of the target node. PDGs were extended to System Dependence Graphs (SDG) to accommodate inter-procedural slicing by Horwitz *et al.* in [60]. A

system dependence graph consists of the program dependence graphs associated with all procedures in the program and additionally the call dependencies between call nodes and called procedures, the parameter-in (parameter-out) dependencies between actual and formal parameters passed to (returned from) each procedure at the call site and transitive dependencies (called summary edges). The partial ordering of the vertices induced by the dependence edges must be preserved in order to guarantee semantic correctness of the program. Several extensions and approximation algorithms were proposed over these approaches to facilitate features such as alias calculation and unstructured program flow. Details can be found in the survey literature. Slicing using SDGs has been evaluated to be the most precise and complete slicing method currently available [123, 55, 13].

4.1.2 Slicing for program comprehension

Frank Tip [123] and Binkley *et al.* [13] outline a number of interesting uses of program slicing. Some of the notable uses of static program slicing are outlined here. Lyle in his PhD thesis [78] outlined the use of program slicing to localise a bug. He called the concept *program dicing*, a method for combining the information obtained from different slices to assist in locating the bug. An interesting consequence of this idea was another method to eliminate *dead code* within the program [9]. Horwitz introduced the concept of *program differencing* using static slicing [58] based on her algorithm of static program slicing presented in [60]. Program differencing is a technique for finding out parts of the new program which has changed from its previous/older counterpart. In [44], the notion of *decomposition slice* was introduced, where a decomposition slice with respect to a variable v is defined as the union of slices with respect to v at the statements that output v and the last statement of the program. The intended use of decomposition slice is in the area of software maintenance where it is suggested that the complement of decomposition constitutes the set of independent statements which may be modified without having any side effects to the variable v. In their seminal paper [49], Gupta, Harrold and Soffa proposed the idea of *regression testing* where only the parts affected by the modification

of a previously tested program are re-tested using program slicing while maintaining the coverage of the original test suite. Another important use of slicing is clone *clone detection* where non-contiguous, reordered, and intertwined clones are detected using isomorphic PDG subgraph matching [66].

Program slicing has been found to facilitate program comprehension and it is this specific use that we are particularly interested in. Binkley and Harman, in their survey [13], reported several research efforts undertaken to link slicing with program comprehension factors. Mark Weiser stated in [134] that programmers, while debugging, mentally traverse backwards from the location of bug (variable and location as slicing criterion) to find the program slice and thus related human comprehension of program to slicing. Lyle in his PhD thesis [78] similarly conjectured that programmers using dicing (intersection of two or more slices) have been observed to find faults faster than their counterparts using traditional methods. Francel and Rugaber in [43] compared the level of comprehension of programmers who used slicing to those who do not and hypothesised that the former had a better understanding of code for debugging than the latter. Ott and Thuss used program slicing based metrics to estimate cohesion among program modules [100]. This estimation, in turn, helped in maintaining program modularity. They created *slice profiles* to aid in visualisation of relationships among different slices generated by a program module and extended Weiser's slicing metrics to build a quantitative approach towards measuring module cohesion (compared to previous qualitative approaches).

Among the newer endeavours, Rilling and Klemola, in [109], proposed an approach to identify comprehension bottlenecks by combining slicing with cognitive complexity metrics. They observed that comprehension aids in digging program artefacts and their relationships using reverse engineering — something an adversary would attempt on an obfuscated program. Their approach to measure cognitive complexity of source code involved defining three types of *identifier densities* and using these densities as slicing criteria to aid in comprehension. David Brinkley and Mark Harman did extensive research on the subject of slicing and its use in comprehension. In [12], they performed a large scale empirical study on a million lines of C code to understand how slicing is effective

in reducing the volume of code to be analysed. They collected more than two million slices to understand how the size of slices were affected by taking the calling contexts into account (and vice-versa). Slicing criteria was selected by taking all executable statements of the program in account (the authors justified it by arguing that slicing on all executable statements "allows the experiment to investigate the question of what a typical slice size might be, for a slice chosen at random"). They observed that taking into account the calling context (by expanding struct fields), backward slices included about 28% of the program and forward slices about 26% of the program. With this result they claimed that program slicing can expect to remove about three quarters of a program. However, ignoring calling contexts led to an overall increase of 50% on the slice sizes. The authors commented on the need for future work to study the limitations of underlying analysis algorithm implementations .

In another empirical study [11], Binkley and Harman investigated the interdependencies among program components effected by predicates in the program. Using slicing they came to the conclusion that predicates in a program rarely depend on all the formal parameters and global variables. Their results also show the differences between formal parameter dependence and global variable dependence. Lastly, Meyers and Binkley, in [89], devised slicing based program cohesion metrics to aid in software intervention. Intervention in their context refers to the task of identifying modules within a program that require reconstruction. They followed up on the work of Ott and Thuss [100] to devise slice-based cohesion metrics that quantify the overall quality of source code and could be used to measure software intervention efforts. For these cohesion metrics, they suggested base-line values which could, in turn, be used to identity degraded program modules.

4.1.3 Slicing Notation

We use the notation

$$slice(P, s, \mathcal{V})$$

to denote a backwards slice of program P from the statement s with respect to the set of variables $\mathcal{V}$. The statement s and the set $\mathcal{V}$ are called the *slicing criterion*. We restrict the set $\mathcal{V}$ to variables which are output — an output variable could be as part of a **printf** statement or passed to another method. We will use the statement **out** to denote an output. It is worth nothing that throughout our design, we will be concerned only with backwards slices from **out** statements. Let us illustrate this with the sum/product example of Figure 4.1. The statement **out**(x) produces the sum of the integers 1 to n, which we write as $sum(1..n)$ and **out**(y) produces the product $prod(1..n)$. In Figure 4.1 we have also indicated the backwards slice from **out**(y) by underlining.

```
int sumprod (int n) {
  int i = 0;
  int x = 0;
  int y = 1;
  while (i < n)
  {
    i++;
    x = x + i;
    y = y * i;          }
  out(x);
  out(y);           }
```

Figure 4.1: Method to calculate the sum and product of the first n positive integers (the backwards slice from **out**(y) is indicated by underlined statements).

Now let us perform a simple obfuscation on the sum/product example by adding some statements involving a dummy variable d. We can use a slicer on this program to produce a backwards slice from **out**(y) with respect to the variable y. The obfuscation and the slice are shown in Figure 4.2. We can see that the statements contained in the slice do not include the dummy variable d (and in fact the slice is exactly the same as the one in Figure 4.1). We obtain a similar result if we perform a backwards slice from **out**(x).

If we apply an obfuscation $\mathcal{O}$ to a program P, we assume for now that the obfuscation does not change the input/output behaviour of the program. In reality however, our obfuscations will impose a refinement "relationship" between the original and obfuscated

```
int sumprodobf (int n) {
  int i = 0;
  int x = 0;
  int y = 1;
  int d = x + y;
  while (i < n)
  {
    i++;
    x = x + i;
    y = y * i;
    d = d * x + i + y;          }
  out(x);
  out(y);          }
```

Figure 4.2: Simple obfuscation of the sum/product example (the backwards slice from **out**(y) is indicated by underlined statements).

variables [35] and so we can write $\mathcal{O}(\mathcal{V})$ to denote the set of variables in $\mathcal{O}(P)$ and the output statement s remains but it may take different arguments. This would be discussed in details in the next chapter.

4.2 Using slicing to define obfuscation

As mentioned earlier, slicing is often used to aid program comprehension. As obfuscation is used to make programs harder to understand we should aim to create obfuscations that make slicing less useful — such obfuscations will be called *slicing obfuscations*. But what do we mean by "less useful"? One measure indicating the usefulness of a slicing technique is the size of slice produced by it. We show below a first attempt at defining a slicing obfuscation is that such an obfuscation creates larger slices.

An obfuscation $\mathcal{O}$ is a *slicing obfuscation* for a program P, an output statement s and set of variables $\mathcal{V}$ if it increases the size of the slice, *i.e.*

$$|slice(P, s, \mathcal{V})| < |slice(\mathcal{O}(P), \mathcal{O}(s), \mathcal{O}(\mathcal{V}))|$$

For our restricted language we could measure the size by counting the number of

assignments, loops, conditional and output statements.

Let us consider the following (trivial) program Q:

$$\begin{aligned} s_1 &: \quad x = 2; \\ s_2 &: \quad y = 3; \\ o_1 &: \quad \textbf{out}(x); \\ o_2 &: \quad \textbf{out}(y); \end{aligned}$$

Then $slice(Q, o_2, \{y\}) = s_2; o_2$ and so just contains two statements.

A simple transformation $\mathcal{O}(Q)$ is:

$$\begin{aligned} s_1 &: \quad x = 2; \\ s_3 &: \quad y = 1; \\ s_4 &: \quad y + +; \\ s_5 &: \quad y + +; \\ o_1 &: \quad \textbf{out}(x); \\ o_2 &: \quad \textbf{out}(y); \end{aligned}$$

and now $slice(\mathcal{O}(Q), o_2, \{y\}) = s_3; s_4; s_5; o_2$ and so we have a bigger slice. We can continue performing this kind of transformation and so we can arbitrarily increase the size of the slice — thus we can never obtain a maximal slice. We could just consider the statements in the slice that came from the original program but our transformations may change every statement of the original program and so this would not be a practical measure.

Our example transformation does not affect any of the statements that are left out of the slice (*i.e.* s_1 and o_1). A more suitable obfuscation would try to increase the size of the slice by including more of the statements that are left behind. We call the statements that are omitted from a slice the *orphaned* statements of a slice. We can define:

$$orphan(P, s, \mathcal{V}) = P \backslash slice(P, s, \mathcal{V})$$

Using this expression, we can define a slicing obfuscation as follows:

Definition 1 (Slicing Obfuscation)**.** An obfuscation $\mathcal{O}$ is a **slicing obfuscation** for a program P, an output statement s and set of variables $\mathcal{V}$ if it decreases the number of orphaned statements, *i.e.*

$$|orphan(P, s, \mathcal{V})| > |orphan(\mathcal{O}(P), \mathcal{O}(s), \mathcal{O}(\mathcal{V}))|$$

For the example Q above, suppose that we perform the following obfuscation, $\mathcal{O}(Q)$:

$$\begin{aligned} s_1 &: \quad x = 2; \\ s_3 &: \quad y = x + 1; \\ o_1 &: \quad \textsf{out}(x); \\ o_2 &: \quad \textsf{out}(y); \end{aligned}$$

Now, $slice(\mathcal{O}(Q), o_2, \{y\}) = s_1; s_3; o_2$ and this obfuscation has decreased the number of orphaned statements as well as increasing the size of the slice.

Now consider the program and the backwards slice for y given in Figure 4.1. We can see that slicing at $\textsf{out}(y)$ leaves us with the following statements:

$$\begin{aligned} &\textsf{int } x = 0; \\ &\ldots \\ &\quad x = x + i; \\ &\ldots \\ &\textsf{out}(x); \end{aligned}$$

According to our slicing definition of obfuscation we would like to reduce the number of orphaned statements.

4.3 Metrics

The goal of slicing obfuscations is to make slicing less effective as a tool for program comprehension. We would now like to be able to introduce metrics for measuring the

obfuscatory strength of our transforms with respect to slicing attacks. Meyers and Binkley [89] studied some slice-based metrics which were proposed as measures of the quality of software. Two of the metrics (*Tightness* and *Coverage*) were originally presented by Weiser [134] and the other two (*MinCoverage* and *MaxCoverage*) were proposed by Ott and Thuss [100]. While discussing these metrics, we will also talk about their measurement in our experimental setup.

We recapitulate from Section 4.1.3 that for each program (method) M in our experiments we will concentrate on variables which are output (for instance, as part of a **printf** statement) and so we have a set of output variables V_O. Our slicing point will be the last output statement for each output variable. For each $v_i \in V_O$ the backwards slice for v_i is denoted by SL_i, SL_{int} is defined to be $\bigcap_i SL_i$ (*i.e.* the intersection of all the slices) and $|\ldots|$ denotes the size. The definitions taken from [89] for the four slice-based metrics that we will use are given below.

Tightness Tightness measures the number of statements in every slice.

$$T(M) = \frac{|SL_{int}|}{|M|}$$

Minimum Coverage Minimum coverage is the ratio of the smallest slice in a method to the method's length.

$$MinC(M) = \frac{1}{|M|} \min_i |SL_i|$$

Coverage Coverage compares the length of slices to the length of the entire method.

$$Cov(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|SL_i|}{|M|}$$

Maximum Coverage Maximum coverage is the ratio of the largest slice in a method to the method's length.

$$MaxC(M) = \frac{1}{|M|} \max_i |SL_i|$$

These metrics give values in the range between 0 and 1 (so we give our values as

percentages) and, in terms of slicing, good slices are indicated by low metric values. Our aim when performing slicing obfuscations is to increase some (and ideally *all*) of the slicing metrics. To compute the metrics for each slice we need to measure the size of the method and well as the size of the slices in a consistent manner.

4.3.1 Choosing a slicer

Mobile code (such as Java byte code or Microsoft's MSIL) is considered more vulnerable to reverse engineering attacks than binary executables. Nevertheless, we have designed our experiments using the constructs of the C language because it was difficult to find a full-fledged working slicer for a language other than C. Experience with using third-party tools for experimentation suggests that most of the tools built as part of academic projects are in their prototypical phase and not well maintained [82]. The only well-known static slicer for Java programs is Indus [65]; again, it is an academic project and is yet to be empirically evaluated for correctness and performance.

CodeSurfer [3] is an exception — it has been widely used since the release of the prototype Wisconsin Program-Slicing Project in 1996 (based on the slicing algorithm by Horwitz *et al.* [60]) and has been extensively evaluated in published literature [11, 12, 89]. CodeSurfer runs algorithms on system dependence graphs (SDGs), an intermediate structure for representing programs [60]. CodeSurfer has a feature which allows the user to manipulate program points (*i.e.* nodes in the SDG) using a *set calculator*. Additionally it has a query interface through which the output graph can be filtered. A set calculator also provides a suite of logical set operations such as union, intersection, and difference [3] . We can use this calculator to compute measurements for slices as well as for methods and we can perform operations on the sets.

Next we derive our metrics from slice-based metrics and bind them to our definition of slicing obfuscations.

4.3.2 Slicing residues

For each $v_i \in V_O$ (the set of output variables) we define:

$$residue(M, v_i) = M \backslash SL_i$$

So the *residue* of a slice is defined to be the set of points that are orphaned. Using this concept, we can refine the definition of a slicing obfuscation as follows:

Definition 2 (Slicing Obfuscation)**.** An obfuscation $\mathcal{O}$ is a **slicing obfuscation** for a program P and a variable v_i if it decreases the number of orphaned points (the size of the residue), *i.e.*

$$|residue(P, v_i)| > |residue(\mathcal{O}(P), \mathcal{O}(v_i))|$$

Using inspiration from the *Tightness*, *MinCoverage*, *Coverage* and *MaxCoverage* metrics given in [100], we can define four *residue metrics* for a method M. We denote RES_{un} to be the union of all the residues, *i.e.*

$$RES_{un} = \bigcup_{i=1}^{|V_O|} residue(M, v_i)$$

In fact, $RES_{un} = M \backslash SL_{int}$.

MinDensity The minimum density is the ratio of the smallest residue in a method to the method length.

$$Min(M) = \frac{1}{|M|} \min_i |residue(M, v_i)|$$

Density Density compares the average residue size to the method size.

$$D(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|residue(M, v_i)|}{|M|}$$

MaxDensity The maximum density is the ratio of the largest residue in a method to the method's length.

$$Max(M) = \frac{1}{|M|} \max_i |residue(M, v_i)|$$

Compactness Compactness measures the total number of orphaned points in relation to the size of the method.

$$C(M) = \frac{|RES_{un}|}{|M|}$$

Comparing our metrics to those in [100], we find that

$$\begin{aligned} Min(M) &= 1 - MaxCoverage(M) \\ D(M) &= 1 - Coverage(M) \\ Max(M) &= 1 - MinCoverage(M) \\ C(M) &= 1 - Tightness(M) \end{aligned}$$

and by the definitions, we see that

$$Min(M) \leq D(M) \leq Max(M) \leq C(M) \tag{4.1}$$

Our aim when obfuscating will be to make these metrics as low as possible (*i.e.* to have as few orphaned points as possible). In Table 4.1 we use our residue metrics to evaluate the effectiveness of some obfuscating transforms on a particular example.

For a particular method M, the size $|M|$ is obtained by computing the number of program points contained within the method itself (and not any that are contained in calls to other methods) . The set of program points for a backwards slice from a point in M may contain points from other methods, such as method calls, and so the size of a slice may actually be bigger than $|M|$. In our experiments, we will only concentrate on a single method M. So, when computing the size of the slice, we will only consider the part of the slice that is contained in M. Using flattening techniques, we inline the code of every method in our examples into one method M which contains the desired slicing

```
original() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar()) != EOF) {
    nc++;
    if (c ==' ' || c ==' \n' || c ==' \t') in = F;
    else if (in == F) {in = T; nw++;}
    if (c ==' \n') nl++;          }
  out(nl, nw, nc);   }
```

Figure 4.3: Method to calculate the number of lines, words and characters in a piece of text (the backwards slice for nl in indicated by underlined points).

criterion (output variables).

4.4 Illustration with an example

Our objective in this section is to illustrate the use of residue metrics introduced earlier to empirically evaluation evaluate some of the slicing obfuscations. To help us to explain how these transformations operate we will use a running Word Count example for the rest of this chapter.

4.4.1 The Word Count Example

The Word Count program takes in a block of text and outputs the number of lines (nl), words (nw) and characters (nc). So, for this example, $V_O = \{nl, nw, nc\}$. The method can be seen in Figure 4.3. Note that we write **out**(nl, nw, nc) as a shorthand for the three **printf** statements contained in the actual method.

Our slicing criteria will be the output statement and one of the three output variables. In Figure 4.3 the underlined points denote the backwards slice from nl given by CodeSurfer. We can see that the residue for nl contains program points for both nc and nw. The aim of our obfuscations will be to create dependencies between the three output variables so as to decrease the sizes of the residues.

We discuss several techniques to decrease the size of the residues for the output variables and thus decrease the effectiveness of slicing. In Table 4.1 we summarise the results

Method M	$\|M\|$	$\|V_O\|$	Residue size				$Min(M)$	$D(M)$	$Max(M)$	$C(M)$
			nl	nw	nc	$\cup$				
original	32	3	21	14	22	26	43.8%	59.4%	68.8%	81.3%
bogus	38	3	8	8	8	9	21.1%	21.1%	21.1%	23.7%
*encode*1	35	3	24	17	7	29	20.0%	45.7%	68.6%	82.9%
*encode*3	42	3	8	8	10	11	19.0%	20.6%	23.8%	26.2%
loop	36	3	7	7	7	8	19.4%	19.4%	19.4%	22.2%
split	44	3	9	9	9	10	20.5%	20.5%	20.5%	22.7%
array	28	3	7	7	7	8	25.0%	25.0%	25.0%	28.6%

Table 4.1: Table of results for the residues of our Word Count example

of our different transformations of the Word Count example. Each row of the table contains the results for a particular experiment where we record the size of the method, the size of each residue (calculated using CodeSurfer by subtracting the slice size from the method size), the size of the union and the results for our four residue metrics from Section 4.3.2. The topmost row of the table displays the measurements for the original Word Count method. The obfuscations (column 1 names) are discussed in the next subsections. The results of Table 4.1 are represented pictorially by the bar chart of Figure 4.4. It can be seen from the bar chart that all obfuscations decrease the residue percentages and it is clear which transformations give the best results.

4.4.2 Adding a bogus predicate

Suppose that the residue for a variable y contains an assignment for another variable x. To include a statement $x = G$ in a slice for y we can transform it to

$$x = G;\ \textbf{if}\ (p^F)\ y = H(x);$$

where p^F is a false predicate and H is an expression depending on x. As we appear to have set up that the definition of y depends on x then the statement $x = G$ will be included in the slice for y. Another possibility is the following transformation:

$$x = G;\ S; \quad \rightsquigarrow \quad x = G;\ \textbf{if}\ (q^T)\ S;\ \textbf{else}\ y = H(x); \tag{4.2}$$

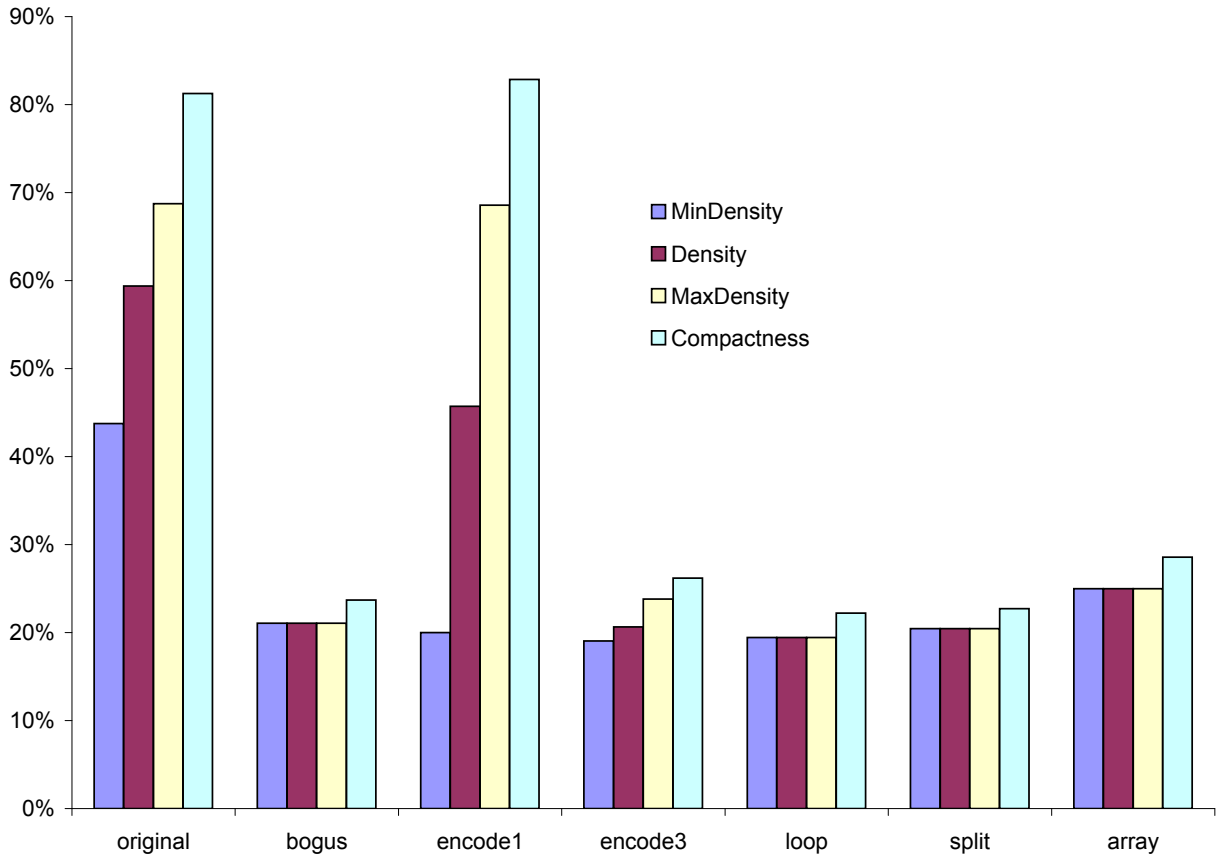

Figure 4.4: Bar Chart showing the residue metrics for our Word Count examples

where S is a statement and q^T is a true predicate.

For Word Count, we added the following bogus predicates

$$\textbf{if } (c ==' \backslash n') \ \{\textbf{if } (nw <= nc)\ nl++;\}$$
$$\textbf{if } (nl > nc)\ nw = nc + nl;$$
$$\textbf{else } \{\textbf{if } (nw > nc)\ nc = nw - nl;\}$$

at the end of the **while** loop. These predicates add dependencies between the three variables and so the slice for each variable contains the definitions for the other variables. In Figure 4.5 we can see this method with the backwards slice from nc. Since nc is incremented for every character and nw and nl are only incremented for certain characters, we have the following invariant

$$nc \geq nw \wedge nc \geq nl \tag{4.3}$$

```
bogus() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar())! = EOF) {
    nc++;
    if (c ==' ' || c ==' \n' || c ==' \t') in = F;
    else if (in == F) {in = T; nw++;}
    if (c ==' \n') {if (nw <= nc) nl++;}
    if (nl > nc) nw = nc + nl;
    else {if (nw > nc) nc = nw − nl;}          }
  out(nl, nw, nc);    }
```

Figure 4.5: Addition of bogus predicates (with the backwards slice for nc).

and so our predicates leave the values of the three output variables unchanged.

In Table 4.1 we can see for the *bogus* method that we increase the method size by 19% but we significantly decrease the size of the residues and thus the slice sizes are increased.

4.4.3 Variable Encoding

In the Chapter 2, we discussed variable encoding obfuscation and here we present an encoding in which a variable x is transformed so that it seems to depend on a variable y:

$$x \rightsquigarrow \alpha * x + \beta * y$$

where α and β are constants.

For Word Count, we will perform the following encoding:

$$nc \rightsquigarrow nc + nl - nw$$

Before nc is output, we need to include the statement

$$nc = nc - nl + nw;$$

since the original value of nc has to be obtained before being output.

The full method for this transformation can be seen in Figure 4.6 with the backwards

```
encode1() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar()) != EOF) {
    nc++;
    if (c ==' ' || c =='\n' || c =='\t') in = F;
    else if (in == F) {in = T; nw++; nc--;}
    if (c =='\n') {nl++; nc++;}          }
  nc = nc - nl + nw;
  out(nl, nw, nc);      }
```

Figure 4.6: Simple variable encoding (with the backwards slice for nc).

```
encode3() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar()) != EOF) {
    nc++; nl++;
    if (c ==' ' || c =='\n' || c =='\t') in = F;
    else if (in == F) {in = T; nw++; nc--; nl--;}
    if (c =='\n') {nl++; nw--;}          }
  int t = nl - nc;
  nc = nc + nw - t;
  nl = t + nc;
  nw = nw + nl - nc;
  nl = nl - nc;
  out(nl, nw, nc);      }
```

Figure 4.7: Three variable encodings (with the backwards slice for nw).

slice for nc indicated. We can see that we have successfully reduced the residue size for nc from 22 to 7 but we fail to do so for the other two outputs variables and, in fact, we increase the size of the residues . This highlights the importance of adding obfuscations for *all* of the output variables since we would ideally want RES_{un} to decrease for a slicing obfuscation to be considered successful (refer to Definition 2).

To create dependencies for *all* the variables, we can perform these three encodings:

$$nc \rightsquigarrow nc - nw \qquad nw \rightsquigarrow nw - nl \qquad nl \rightsquigarrow nl + nc$$

We apply these transformations in order starting with the one for nc. Before each output statement, we restore back the original values of the variables.

```
loop() {
  int c, nl = 0, nw = 0, nc = 0, in, j = 0;
  in = F;
  while (((c = getchar()) != EOF) && (j >= 0)) {
    nc++;
    if (c ==' ' || c =='\n' || c =='\t') in = F;
    else if (in == F) {in = T; nw++;}
    if (c =='\n') nl++;
    j = nc + nl - nw;          }
  out(nl, nw, nc);   }
```

Figure 4.8: Adding a new loop variable (with backwards slice from nw)

After performing these encodings, in order, we obtain the method given in Figure 4.7. Note that we have had to use a temporary variable t (which is included in the slices) and, from Table 4.1, we can see that for *encode*3 the size has increased by 31%. However, we have reduced the sizes of the three residues and so the metrics values have significantly decreased.

4.4.4 Adding to the guard of a **while** loop

For Word Count, we add a new, fresh variable j to the loop with which we can create dependencies on the three output variables. Let us suppose that we add the statement $j = nc + nl - nw$ into the loop. Before the loop, we initialise j by adding the statement **int** $j = 0$. By our invariant, Equation (4.3), we can see that the value of j is always non-negative in the loop and so we change the loop header to

$$\textbf{while } (((c = \textbf{getchar}()) \,!= \textbf{EOF}) \;\&\&\; (j >= 0))$$

We then add our assignment for j into the loop. The full method can be seen in Figure 4.8. From Table 4.1, for *loop* we have only added 4 extra points to the method but the size of the residues are all decreased (for example, a 68% decrease for nc).

The addition of the guard does not change the **while** loop (as j is always non-negative) and the extra statement for j does not change the values of nl, nw or nc. Thus the transformation is correct.

```
split(){
  int c, nl = 0, nw = 0, a = 0, b = 0, in;
  in = F;
  while ((c = getchar()) != EOF) {
    if (b == 9) {a ++; b = 0;} else {b ++;}
    if (c ==' ' || c ==' \n' || c ==' \t') in = F;
    else if (in == F) {
      if (b < 10) {in = T; nw ++;}
      else {nw = nw + nl; b = b + nw;}   }
    if (c ==' \n') {nl ++; if (in == T) {nl = a + nl;}}
  out(nl, nw, 10 * a + b);   }
```

Figure 4.9: Variable Split (with the backwards slice for $10 * a + b$).

4.4.5 Using a Variable Split

A variable v can be split up to two or more variables, which we call the *split components*. The information contained in v is split across the components and since we know the relationship between the original variable and its components we can create invariants based on the split components.

So for Word Count, we will split nc into two other integers by using $nc \rightsquigarrow \langle nc\,/\,10,\ nc\,\%\,10\rangle$ and let $a = nc\,/\,10$ and $b = nc\,\%\,10$ — we have that $0 \leq b \leq 9$ which we take to be our invariant.

The first step is apply the transformation to nc. The statement $nc = 0$ becomes $a = 0;\ b = 0;$ and the output for nc is now **out**$(10 * a + b)$. To measure the size of the slice (and the residue) we will take backwards slice of $10 * a + b$ and in Table 4.1 the values for nc represents the values for the slice of a and b.

In Figure 4.9 we can see the method after this transformation. In the method, we have added two bogus predicates as the variable split in isolation does not produce a very good slicing obfuscation. The first predicate uses our invariant on b to add in dependencies for nw and b. The other comes after the increment for nl where we add in a dependency on nl. From Table 4.1 we can see that for *split* we have reduced the residue sizes for the three output variables but we increased the method size by 38%. This size increase is due to the extra assignments and predicates needed for this transformation.

```
array() {
  int c, in;
  int a[3] = {0, 0, 0};
  in = F;
  while ((c = getchar()) != EOF) {
    a[2] ++;
    if (c == ' ' || c == '\n' || c == '\t') in = F;
    else if (in == F) {in = T; a[1] ++;}
    if (c == '\n') a[0] ++;          }
  out(a[0], a[1], a[2]);    }
```

Figure 4.10: Transformation to arrays (with the backwards slice for $a[0]$).

4.4.6 Arrays

So far we have only considered simple variable transformations but what would happen to the slices (and the obfuscations) if we use arrays? Suppose that we had an expression of the form:

$$x = f(a[0])$$

for some variable x and array a. If we perform a backwards slice for x from this point then the slice for x will contain assignments for other array indices and not just for $a[0]$.

For Word Count, we can perform this transformation:

$$nl \rightsquigarrow a[0] \qquad nw \rightsquigarrow a[1] \qquad nc \rightsquigarrow a[2]$$

This transformation actually is just a variable renaming. If we ensure that the array a is indexed by using only 0, 1 and 2 then the transformation is correct. In Figure 4.10 we can the result of taking backwards slice for $a[0]$ (*i.e.* nl). The results for *array* can be seen in Table 4.1 where the results for nl, nw and nc represent $a[0]$, $a[1]$ and $a[2]$ respectively. The size of the new method is actually smaller than the *original* method — this is due to the initialisation of the variables. This simple transformation results in significantly decreasing the residues for all three of the output variables.

Once we use arrays we can employ many different array transformations. However, array restructuring transformations such as splitting and folding [27], which are often used

as array obfuscations, are not very suitable for creating dependencies on other variables because conservative points-to analysis algorithms do not in general distinguish between array indices.

4.5 Further Examples

In the previous section we gave details of how to produce slicing obfuscations for the Word Count program. We now use some of some of the data and control-flow obfuscations given in Section 4.4 to transform four more C programs. In the rest of this section we briefly describe each of the programs, the obfuscations that we performed and the slice sizes we obtained using CodeSurfer. In the final section (Section 4.5.5) we summarise the results of our experiments which can be seen in Table 4.2. Each row of the table contains the results for a particular method: we have recorded the size of the method, the size of each slice with respect to a particular variable and the size of the intersection of each of the slices (all of these values were computed using the set calculator of CodeSurfer). We have computed the values for the four slicing metrics (mentioned in Section 4.3) as percentages.

4.5.1 Product and Sum

Using the example we started this chapter with, a method calculating the product (*prod*) and sum (*sum*) of the first n positive integers was considered next. Our program contained a **while** loop using a variable i which counts up from 0 to n.

We first created a dependency for *prod* on *sum* by adding the condition of $\vee sum < 0$ in the guard of the **while** loop. Then we added a bogus predicate p^T around the statement $prod = prod * i$. The metrics values for three methods can be seen in Table 4.2. The method *ps* relates to the original obfuscated method, in $psObf1$ the loop dependency was created and finally we created the bogus predicate in $psObf2$. Details of some more slicing obfuscations for the product and sum program can be found in [40].

Method M	$\|M\|$	$\|V_O\|$	For each v_i, the slice size $\|SL_i\|$						$\|SL_{int}\|$	$T(M)$	$MinC(M)$	$Cov(M)$	$MaxC(M)$
ps	21	2	$prod$	12	sum	12			7	33.3%	57.1%	57.1%	57.1%
$psObf1$	22	2	$prod$	16	sum	13			11	50.0%	59.1%	65.9%	72.7%
$psObf2$	26	2	$prod$	19	sum	19			17	65.4%	73.1%	73.1%	73.1%
$search$	107	2	n	9	$secs$	11			2	1.9%	8.4%	9.3%	10.3%
$searchObf1$	120	2	n	45	$secs$	11			10	8.3%	9.2%	23.3%	37.5%
$searchObf2$	127	2	n	49	$secs$	48			46	36.2%	37.8%	38.2%	38.6%
rov	124	2	$fuel$	23	$dist$	46			19	15.3%	18.5%	27.8%	37.1%
$rovObf1$	129	2	$fuel$	60	$dist$	46			45	34.9%	35.7%	41.1%	46.5%
$rovObf2$	132	2	$fuel$	62	$dist$	60			59	44.7%	45.5%	46.2%	47.0%
$rovObf2np$	94	2	$fuel$	62	$dist$	60			59	62.8%	63.8%	64.9%	66.0%
$scatter$	143	3	si	116	ru	111	i	9	8	5.6%	6.3%	55.0%	81.1%
$scatterObf1$	148	3	si	132	ru	132	i	132	131	88.5%	89.2%	89.2%	89.2%
$scatterObf2$	150	3	si	139	ru	139	i	139	138	92.0%	92.7%	92.7%	92.7%

Table 4.2: Table showing the slicing metrics values for four example programs. There is a separate row in the table for recording the slicing metric values of the example programs and their obfuscated counterparts. The row labelled ps, for example, records the slicing metric values for the unobfuscated instance of Product Sum example; whereas, $psObf1$ indicates the metric values when slicing obfuscations have been applied. The columns from $|M|$ to $|SL_{int}|$ reflect measures with respect to the number of SDG nodes. The latter columns indicate the slicing metric values.

4.5.2 Search Sort

The search sort program (obtained from [129]) takes an argument n from the user, performs different sorts and searches on n elements and then displays the time taken to do each one. So the two output variables at each stage are the number of elements n and the number of seconds *secs*. This program is different from those considered so far as it contains various methods which the main method calls. However the results for all of these methods are discarded and only the time taken to perform each method is computed. Thus slices for both n and *secs* only contain statements from the main method. So for our experiments we only consider changing statements in the main method. The metric results for the main method *search* can be found in Table 4.2.

As n is constant throughout the program (it is inputted by the user) we attempt to make n vary by adding a variable encoding. We would like to create a dependency for n on *sec* but n is declared as an integer and *secs* is a float. So we declare a new integer variable k which we define as $k = (\textbf{int})\ 10 * secs$ and we perform the transformation $n \rightsquigarrow n + k$. By the replacement rules for variable encoding we need to redefine k every time that *secs* is redefined and so we can use a different declaration for k. Note that this kind of transformation could make the value of n overflow and so we should put in checks to ensure that this does not happen. The results for this obfuscation can be found in Table 4.2 for the method $searchObf1$ and we can see that we have significantly increased the slice size for i but the size for *secs* is unchanged.

To create some dependencies for *secs* we place two bogus predicates near the end of the method. The value of *secs* is obtained in the following way:

$$c1 = \textbf{clock()};$$
$$search\_method();$$
$$c2 = \textbf{clock()};$$
$$secs = c2 - c1;$$

So, for example, we can change the first assignment to:

$$\textbf{if } (secs >= 0)\ c1 = \textbf{clock()};\ \textbf{else } c1 = bogus(n);$$

The results after the addition of these predicates can be found in Table 4.2 for the method $searchObf2$ and we can see that we have increased the slice size for $secs$.

4.5.3 Rover

The rover program checks whether a plan for manoeuvring a vehicle around an obstacle to a given target, with a limited amount of fuel, satisfies certain constraints. It simulates a land rover vehicle which needs to be driven around a rock on the Martian surface by giving it coordinates as input [129]. However, the vehicle goes off-track from the specified target and it has limited amount of fuel. The challenge is to bring the vehicle within 5km of the target by inputting a sequence of coordinates which will also not exhaust the fuel before it reaches the target. Here we consider two output variables $fuel$ and $dist$. The results for the original method can be seen in the rov row of Table 4.2.

So that $fuel$ depends on $dist$ we perform the following variable encoding $fuel \rightsquigarrow fuel + dist$. Note that we have to add an extra temporary variable t to perform the encoding. The values for this obfuscation can be found in the $rovObf1$ row of Table 4.2.

The value of $dist$ is computed as follows

$$dist = \sqrt{dx^2 + dy^2}$$

where dx and dy are two other variables. To create a dependency for $dist$ we changed an assignment to dx as follows:

$$dx = E;\ \rightsquigarrow\ \textbf{if } (fuel >= dist)\ dx = E;\ \textbf{else } dx = bogus;$$

By the variable encoding above we know that since $dist \geq 0$ then $fuel \geq dist$. The row $rovObf2$ in Table 4.2 contains the values for this obfuscation.

The *rover* example contains many **printf** statements which cannot be contained in any slice. So to find out a more accurate evaluation of our obfuscation we removed all but the final **printf** statements and the result for this method can be found in the $rovObf2np$ in Table 4.2.

4.5.4 Scattering

The scattering program is a typical physics problem dealing with the famous Rutherford's scattering experiment for finding the size of the nucleus of an atom (from Tao Pang's book "An Introduction to Computational Physics" [104]). The original program contained several procedures for performing integration using Simpson's rule, finding roots using the Secant method, and finding the first and second order derivatives with the three-point formula. We flattened the procedures by inlining them in the main() procedure (which we call *scatter*) and we consider three output variables si, ru, and i.

We observe in the program source code that the statement $b = b0 + i * db$; maintains the invariant $i \geq b \ \wedge \ b0 \leq b$. Our first obfuscation consists of performing the following transformations by introducing bogus predicates:

$$si = log(sig[i]); \ \rightsquigarrow \ \textbf{if} \ (i + 1 > b) \ si = log(sig[i]); \ \textbf{else} \ si = bogus;$$

$$ru = log(ruth); \ \rightsquigarrow \ \textbf{if} \ (b0 <= b) \ ru = log(ruth); \ \textbf{else} \ ru = bogus;$$

in the code. The row $scatterObf1$ in Table 4.2 contains the values for this obfuscation. Interestingly, with the addition of 5 more SDG nodes, the slice sizes for variables si, ru and i increases from 116 nodes, 111 nodes and 9 nodes respectively to 132 nodes. In an attempt to provide maximum code coverage, we introduce another similar bogus predicate to include an additional 7 nodes in each of the slices of si, ru, and i. The row $scatterObf2$ in Table 4.2 indicates the changes brought about by this obfuscating transform.

4.5.5 Results and discussion

In Table 4.2 we gave the results for our experiments using the slicing metrics mentioned in Section 4.3. We can easily see that, for each obfuscation, we have increased the values of *all* the slicing metrics and we can also see that we do not significantly increase the size of the methods (the worst is a 24% increase for ps but the size of the slices increase by 58%). The metric values for the search sort obfuscations are generally much lower than those for the other programs. This is because the search sort program uses different searching and sorting algorithms contained within other methods and we have only considered intra-procedural slices. We decided to flatten the scattering program before obfuscating and so we were able to obfuscate the program more successfully.

Using the results of Table 4.2 we can compute the values for our residue metrics (given in Section 4.3.2) and these values can be seen in Table 4.3. We can see that we have managed to reduce the residue metrics values which was our stated aim. One aspect that is key to decreasing the residue metrics values is to ensure that we decrease $|RES_{un}|$ (*i.e.* the union of all the residues). Decreasing the size of the union of the residues means that we decrease the value of the compactness metrics and, by Equation (4.1), we are likely to decrease the values of the other residue metrics. To reduce the union of the residues (which is equivalent to increasing the intersection of the slices), we need to add dependencies between all of the variable in V_O.

The result of Table 4.3 is represented graphically using a bar chart in Figure 4.11. This figure is an extrapolation of the information represented in Figure 4.4 and it groups the residue metrics for each of our example programs (12 programs grouped in 4 categories). We can see from Figure 4.11, a better slicing obfuscation reduces the percentage of the residues that are left behind after slicing.

As pointed out in Chapter 2, the objective of obfuscation is not in guaranteeing an absolute protection of software against all forms of reverse engineering attacks but in providing a time-limited security against a well defined adversary. No concrete definition of deobfuscation (or what constitutes “cracking” of an obfuscation) exists in the literature

Method M	$\|M\|$	$\|V_O\|$	For each v_i, the residue size $\|RES_i\|$						$\|RES_{un}\|$	$Min(M)$	$D(M)$	$Max(M)$	$C(M)$
ps	21	2	$prod$	9	sum	9			14	42.9%	42.9%	42.9%	66.7%
$psObf1$	22	2	$prod$	6	sum	9			11	27.3%	34.1%	40.9%	50.0%
$psObf2$	26	2	$prod$	7	sum	7			9	26.9%	26.9%	26.9%	34.6%
$search$	107	2	n	98	$secs$	96			105	89.7%	90.7%	91.6%	98.1%
$searchObf1$	120	2	n	75	$secs$	109			110	62.5%	76.7%	90.8%	91.7%
$searchObf2$	127	2	n	78	$secs$	79			81	61.4%	61.8%	62.2%	63.8%
rov	124	2	$fuel$	101	$dist$	78			105	62.9%	72.2%	81.5%	84.7%
$rovObf1$	129	2	$fuel$	69	$dist$	83			84	53.5%	58.9%	64.3%	65.1%
$rovObf2$	132	2	$fuel$	70	$dist$	72			73	53.0%	53.8%	54.5%	55.3%
$rovObf2np$	94	2	$fuel$	32	$dist$	34			35	34.0%	35.1%	36.2%	37.2%
$scatter$	143	3	si	27	ru	32	i	134	135	18.9%	45.0%	93.7%	94.4%
$scatterObf1$	148	3	si	16	ru	16	i	16	17	10.8%	10.8%	10.8%	11.5%
$scatterObf2$	150	3	si	11	ru	11	i	11	12	7.3%	7.3%	7.3%	8.0%

Table 4.3: Table of residue metrics values for four example C programs

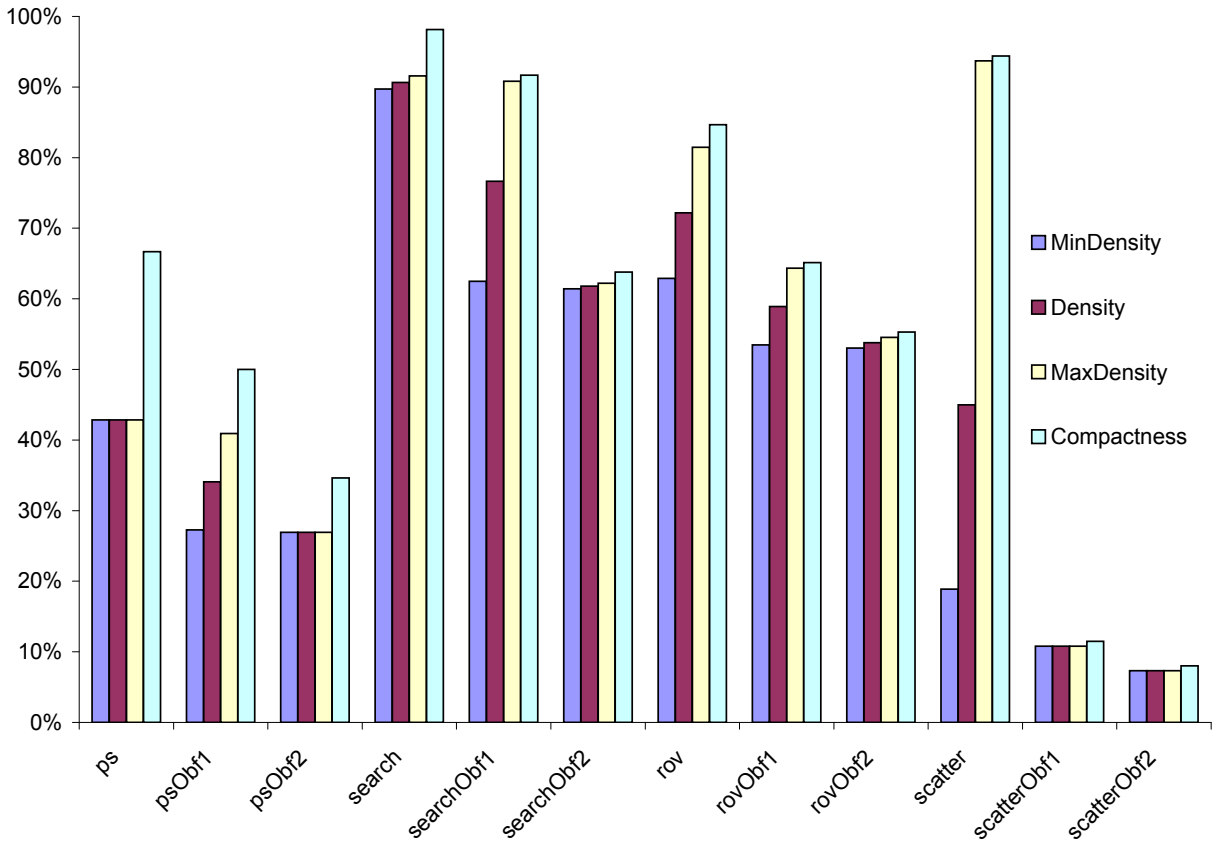

Figure 4.11: Bar Chart showing residue metric values for all our example programs

and in Chapter 3, we attempted to link deobfuscation with reverse engineering of code using program analysis tools. Furthermore, at the start of this chapter, we cited evidence in existing literature from the software engineering community that considers slicer to be a potent program comprehension tool. We also cited previous work that used slicing metrics to evaluate the effectiveness of slicing (with respect to the degree of comprehensibility it attains). Residue metrics, which are derived from slicing metrics, measure the degree of code incomprehensibility. Therefore, the rationale behind using residue metrics is to evaluate the degree to which obfuscations, that are designed from inherent program dependencies, could hinder code comprehension attacks using a static program slicer. It would be interesting in the future to see if human factor experiments such as the one conducted by Nakamura *et al.* [94] corroborates with residue metrics. With respect to such an experiment, it would be interesting to know whether obfuscated programs with small residue metrics are more difficult to comprehend (using the Virtual Mental Simulation Model) than their obfuscated counterparts.

It is also worth noting the obfuscations were applied on a small test suite of programs manually since we lack a context-sensitive automatic obfuscator. With respect to obfuscation, context sensitivity means that an automatic obfuscator should be able to determine the category of obfuscations to apply by inspecting the "nature" of a program. As stated before, a system program that is a collection of library calls is not a good candidate for data obfuscation since it lacks sufficient amount of variables to encode, merge, or split. Current obfuscators such as SandMark [96] and DashO [61] apply all obfuscations to all programs resulting in obfuscated programs that are non-stealthy (since obfuscating transforms "stand out" from rest of the code) and are prone to slicing attack. A future direction of research would be develop heuristics that take into account the context of code before applying obfuscations.

4.6 Conclusion

A contribution of this chapter is in justifying the use of static slicer as an attack tool and introducing the definition of "slicing obfuscations" by considering the statements that were left behind after slicing — which are called *orphaned* statements. Our ideal obfuscation for a particular program will minimise the number of orphaned statements for all output variables. Furthermore, we have proposed a new measurement for slicing obfuscations called a *residue* which consists of points which are *orphaned* (*i.e.* left behind) by slicing. Our goal has been to reduce the number of orphaned points. We derived residue metrics based on existing slicing metrics that indicate the degree of incomprehensibility added to by a slicing obfuscation. In Table 4.1 and Table 4.3 we saw that, according to our residue metrics, we have successfully created transformations to reduce the size of residues. This in comparison to [81] is a better measure of the obfuscatory strength of slicing obfuscations. Since our metrics are related to the slicing based metrics from [100], our obfuscations appear to make slicing less useful. The results for the single variable encoding *encode*1 for the Word Count example highlight the importance of ensuring that when obfuscating we consider the slices for all variables.

We have only conducted relatively small experiments but it has shown us how to use obfuscations to decrease the effectiveness of slicing. Moreover, in our experiments we only used simple obfuscations to illustrate the techniques used to create slicing obfuscations. Even with these simple transformations we have still managed to decrease the effectiveness of slicing which was our stated goal. When faced with an attacker who is armed with more than just a slicer we will obviously have to design more complicated transformations. This will involve creating predicates that are harder for an attacker to understand, using different program constructs such as pointers and dealing with inter-procedural constructs.

It is worth noting that we have intentionally not considered an equivalent residue metric for the slicing metric *overlap*. Overlap is based on the number of statements common to all slices.

Overlap Overlap is the ratio of non-unique to unique statements in each slice.

$$Overlap(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|SL_{int}|}{|SL_i|}$$

An equivalent metric

$$\frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|RES_{un}|}{|residue(v_i)|}$$

would give a value ≥ 1 since $|RES_{un}| \geq |residue(v_i)|$ and hence is not suitable. Similarly a metric such as the following is another possible alternative.

$$1 - \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|residue(v_i)|}{|RES_{un}|} \equiv \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|RES_{un}| - |residue(v_i)|}{|RES_{un}|}$$

This metric measures the number of residues in union that are not the result of a particular slice. But this metric does not correlate well with a method that has been more obfuscated since it merely measures how partitioned the residues are. So, a low value indicates that the residues are not well partitioned as size of residues is almost the size of union. A high value indicates that the residues are well partitioned. Finding new metrics that correlate with the degree of obfuscation is a future research direction.

5

Proving Slicing Obfuscations Correct

After introducing slicing obfuscations and empirically evaluating their obfuscatory strength with a static slicer in the last chapter, we attempt to prove them correct using an existing proof framework originally developed by Drape in his DPhil thesis [36]. The contribution of this chapter lies in demonstrating that Drape's framework can be used to prove imperative transformations, such as slicing obfuscations, correct. Also, interesting questions of composition and placement of obfuscating transforms are addressed in this chapter.

We start by recapitulating Drape's proof framework and then outline some proof rules for proving imperative transforms correct. This is followed by proofs of correctness for all the obfuscating transforms used in the previous chapter. We conclude the chapter with a section on applying the transforms and some interesting open issues.

5.1 Proof Framework

In [36] a framework for proving the correctness of obfuscations for abstract data-types using functional refinement [35] was given . In this section, we recapitulate the proof framework.

Suppose that a data-type D is obfuscated using an obfuscation $\mathcal{O}$ to produce a data-type E. Under this framework, an abstraction function $af :: E \rightarrow D$ and a data-type invariant dti are needed such that, for $x :: D$ and $y :: E$:

$$x \rightsquigarrow y \iff (x = af(y)) \wedge dti(y) \tag{5.1}$$

The term $x \rightsquigarrow y$ is read as "x is obfuscated by y". This is the standard definition for *data refinement* where a data-type that is being data-refined is called *abstract* and the target data-type is called *concrete*. A variable that is an instance of these data-types are called an abstract and concrete variable. In Equation 5.1, x is an abstract variable and y is the concrete variable. In the data refinement literature, a *coupling invariant* CI is a Boolean expression that relates the abstract and concrete variables [91]. When a CI expresses an abstract variable as a function of the concrete variable, the function is called abstraction function af. A data-type invariant dti is an additional constraint that is used in conjunction with af to restrict a data-type to contain only those values that satisfy the Boolean expression. It is imposed only on concrete data-type and not on the abstract one since since the constraint is needed to guarantee that for many possible concrete values, there is only one distinct value in the abstract domain. Morgan in [91] cites an example for representation of sets by sequences to demonstrate the use of dti. In a sequence, elements may appear in a different order and/or duplicated; however, many distinct sequences correspond to at most one set. For this example, an af makes a set from a sequence. A dti can be used to further impose a constraint on the refinement — for example, the sequences are kept in order or that they do not contain duplicated elements.

Drape in his thesis observed that in order to obfuscate a program, either its algorithm

or its data structures can be obfuscated. This proof framework concentrates on the latter and considers obfuscation as a data-refinement on data-types. For a function $f :: D \to D$, an obfuscated function $\mathcal{O}(f)$ is correct with respect to f if it satisfies:

$$(\forall x :: D;\ y :: E)\ x \rightsquigarrow y \Rightarrow f(x) \rightsquigarrow \mathcal{O}(f)(y)$$

Using Equation (5.1) we can rewrite this as

$$f \cdot af = af \cdot \mathcal{O}(f) \tag{5.2}$$

If we have a conversion function $cf :: D \to E$ that satisfies $af \cdot cf = id$ then we can rewrite Equation (5.2) to obtain:

$$f = af \cdot \mathcal{O}(f) \cdot cf$$

and we can use this equation to prove the correctness of $\mathcal{O}(f)$.

In this contribution we will concentrate on proving slicing obfuscations, which are imperative program constructs, correct. In the remainder of this section, we discuss how this functional proof framework can be adapted in the imperative program domain[1].

5.1.1 Modelling statements as functions

We model *statements* to be functions on states and so a statement has the following type: $statement :: state \to state$ where a *state* is defined to be a set of mappings from variables to values (or expressions computing values). We assume that the variables are integer valued and all expressions consist of arbitrary-precision arithmetic operators. We concentrate on code fragments with no methods, exceptions or pointers.

Suppose that we have a set of states $\mathcal{S}$. For some initial state $\sigma_0 \in \mathcal{S}$ and some statement T, the effect of statement T on σ_0 is to produce a new state $\sigma_1 \in \mathcal{S}$ such that $\sigma_1 = T(\sigma_0)$. Suppose that we have a sequential composition (;) of statements, which we will call a *block*, $B = T_1; T_2; \ldots; T_n$. If the initial state is σ_0 then the final state σ_n is given

[1]This adaptation itself is not a contribution of this thesis and is attributed to Drape

by

$$\sigma_n = B(\sigma_0) = T_n\left(\ldots T_2\left(T_1\left(\sigma_0\right)\right)\ldots\right)$$

For our simple language, we consider the following statement types: *skip*, *assignments* ($var = expr$), *conditionals* (**if** *pred* **then** *stats* **else** *stats*) and *loops* (**while** *pred* **do** *stats*).

The statement *skip* does not change the state and so if $S \equiv skip$ then $S(\sigma_0) = \sigma_0$. For an assignment A of the form $A \equiv x = e$, if the initial state σ_0 contains the mapping $x \mapsto x_0$ then the state after the assignment can be written as

$$A(\sigma_0) = \sigma_0 \oplus \{x \mapsto e[x_0/x]\} \tag{5.3}$$

using functional overriding ($\oplus$) and substitution ($/$).

Now suppose we have conditional statement C which has the form

$$C \equiv \textbf{if } p \textbf{ then } T \textbf{ else } E$$

where p is a predicate with type $p :: state \to \mathbb{B}$ and T and E are blocks. Then for some initial state σ_0 we have that

$$C(\sigma_0) = \begin{cases} T(\sigma_0) & \text{if } p(\sigma_0) \\ E(\sigma_0) & \text{otherwise} \end{cases} \tag{5.4}$$

A loop statement L has the form $L \equiv$ **while** p **do** S where p is a predicate and S is a block. Then for some initial state σ_0 we have that

$$L(\sigma_0) = S^i(\sigma_0) \text{ where } i = \min\{n :: \mathbb{N} \mid p\left(S^n(\sigma_0)\right) = False\} \tag{5.5}$$

Note that this minimum does not exist if the loop fails to terminate.

5.1.2 Using the refinement framework

We suppose that a data obfuscation acts on a state σ to produce a new state $\mathcal{O}(\sigma)$ and so we can consider the set of states $\mathcal{S}$ to be obfuscated to produce a new set of states $\mathcal{O}(\mathcal{S})$. To specify a data obfuscation $\mathcal{O}$ we will supply two functions, an abstraction function

$$af :: state \rightarrow state$$

such that

$$af(\gamma) = \delta \Leftrightarrow \gamma \in \mathcal{O}(\mathcal{S}) \wedge \delta \in \mathcal{S}$$

and a conversion function

$$cf :: state \rightarrow state$$

which is a pre sequential inverse for af. A pre sequential inverse same as left inverse when defined with respect to a set, a binary relation and an identity element. Similarly, af is the post sequential (right inverse) of cf, *i.e.*

$$cf; af \equiv skip. \tag{5.6}$$

As well as these functions, for refinement, we require an invariant I on the obfuscated state such that for states $\sigma :: \mathcal{S}$ and $\mathcal{O}(\sigma) :: \mathcal{O}(\mathcal{S})$

$$\sigma \rightsquigarrow \mathcal{O}(\sigma) \Leftrightarrow (\sigma = af(\mathcal{O}(\sigma))) \wedge I(\mathcal{O}(\sigma))$$

The expression "$\sigma \rightsquigarrow \mathcal{O}(\sigma)$" means that the state σ is obfuscated (refined) by $\mathcal{O}(\sigma)$. Using the conversion function we have that

$$cf(\sigma) = \mathcal{O}(\sigma) \Rightarrow \sigma \rightsquigarrow \mathcal{O}(\sigma)$$

Note that for most of our transformations unless otherwise stated $I \equiv True$. The type of the abstraction and conversion functions are the same as the type of a statement and so

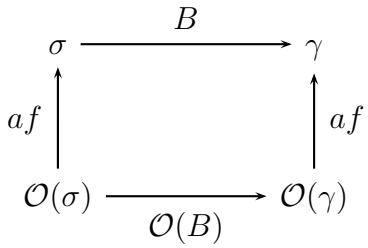

Figure 5.1: A commuting diagram for data obfuscation

we can consider af and cf to be statements (usually assignments) as well as functions. Since our statements are functions on the state, we can produce correctness equations similar to those for functional programs.

Suppose that we have a block B and we want to obfuscate it using data refinement to obtain a block $\mathcal{O}(B)$. We say that $\mathcal{O}(B)$ is *correct* (with respect to B) if it satisfies

$$(\forall \sigma :: \mathcal{S}; \mathcal{O}(\sigma) :: \mathcal{O}(\mathcal{S})) \bullet \sigma \rightsquigarrow \mathcal{O}(\sigma) \Rightarrow B(\sigma) \rightsquigarrow \mathcal{O}(B)(\mathcal{O}(\sigma)) \tag{5.7}$$

We can draw the commuting diagram [35] in Figure 5.1 representing our obfuscation and we obtain the following equation:

$$af; B \equiv \mathcal{O}(B); af \tag{5.8}$$

this corresponds with Equation (5.2). By writing the abstraction function as a statement we can construct two blocks $af; B$ and $\mathcal{O}(B); af$ and proving the equivalence of these blocks establishes that $\mathcal{O}(B)$ is correct. Using the conversion function we can obtain another correctness equation:

$$B \equiv cf;\ \mathcal{O}(B);\ af \tag{5.9}$$

We can extend the refinement notation $\rightsquigarrow$ to statements and blocks. The expression

$$B \rightsquigarrow \mathcal{O}(B) \quad \text{for some } af \text{ (and } cf)$$

says that B is obfuscated by $\mathcal{O}(B)$ and that Equation (5.7) holds.

5.1.3 Obfuscating Statements

Suppose that we have a data obfuscation that changes a variable x using an abstraction function af and a conversion function cf satisfying $cf;\ af \equiv skip$. Then af and cf can be written as statements of the form:

$$af \equiv x = G(x) \qquad cf \equiv x = F(x)$$

for some functions F and G.

Suppose we have an obfuscation for x (with af and cf defined as above) then let us consider the statement $P_1 \equiv x = E$ where E is an expression that may contain an occurrence of x. It can be obfuscated as follows:

$$\mathcal{O}(x = E) \equiv x = F(E') \text{ where } E' = E[G(x)/x] \tag{5.10}$$

For example, the expression $x = x + 1$ would be transformed to $x = F(G(x) + 1)$. Note that the expression $E[G(x)/x]$ denotes how a use of x is obfuscated.

Now let us suppose that $P_2 \equiv \textbf{if } p \textbf{ then } T \textbf{ else } E$ for some predicate p and blocks T and E. We propose that

$$\mathcal{O}(P_2) \equiv \textbf{if } p[G(x)/x] \textbf{ then } \{af; T; cf\} \textbf{ else } \{af; E; cf\} \tag{5.11}$$

with af as above. Using Equation (5.8) we can show that $\mathcal{O}(P_2);\ af \equiv af;\ P_2$.

Finally, suppose that $P_3 \equiv \textbf{while } p \textbf{ do } S$ then, with af as above, we propose that

$$\mathcal{O}(P_3) \equiv \textbf{while } p[G(x)/x] \textbf{ do } \{af; S; cf\} \tag{5.12}$$

and we can use Equation (5.8) to prove that this is correct.

Suppose that s_1 and s_2 are two statements and let us consider how to obfuscate the

sequential composition of these two statements . By Equation (5.8),

$$af;\ s_1 \equiv \mathcal{O}(s_1);\ af \quad \text{and} \quad af;\ s_2 \equiv \mathcal{O}(s_2);\ af$$

We propose that

$$\mathcal{O}(s_1;\ s_2) \equiv \mathcal{O}(s_1);\ \mathcal{O}(s_2) \tag{5.13}$$

Proof. Consider

$$\begin{aligned}
& \mathcal{O}(s_1);\ \mathcal{O}(s_2);\ af \\
\equiv \quad & \{\text{Equation (5.8) for } s_2\} \\
& \mathcal{O}(s_1);\ af;\ s_2 \\
\equiv \quad & \{\text{Equation (5.8) for } s_1\} \\
& af;\ s_1;\ s_2 \\
\equiv \quad & \{\text{Equation (5.8) for } s_1;\ s_2\} \\
& \mathcal{O}(s_1;\ s_2);\ af
\end{aligned}$$

and thus

$$\mathcal{O}(s_1;\ s_2) \equiv \mathcal{O}(s_1);\ \mathcal{O}(s_2)$$

□

So when applying a data obfuscation to a sequence of statements (blocks) we can obfuscate each statement (block) individually and compose the results. That is, for blocks B_1 and B_2, we have:

$$\mathcal{O}(B_1;\ B_2) \equiv \mathcal{O}(B_1);\ \mathcal{O}(B_2)$$

5.1.4 Simultaneous Equations

Suppose that we obfuscate S to obtain $\mathcal{O}(S)$ with abstraction and conversion functions af and cf for the obfuscation. We can use Equation (5.9) to prove that $\mathcal{O}(S)$ is a correct obfuscation of S by showing that the sequence of statements cf; $\mathcal{O}(S)$; af is equivalent to S. Suppose that we have an obfuscation that transforms a variable x using three functions f, g and u (say) as follows:

$$x = f(x); \quad x = u(x); \quad x = g(x)$$

where f, g and u are functions. To simplify this expression we can substitute values of x in sequential order by rewriting the sequence of statements as a set of simultaneous equations. Each definition of a variable will have a different name which is usually the name of the variable with a subscript (*e.g.* x_2) and we will use the convention that the initial value of a variable has a subscript 0. All the uses of a variable are renamed to correspond to the appropriate assignment.

The sequence above can be rewritten as the following set of equations:

$$x_1 = f(x_0); \quad x_2 = u(x_1); \quad x_3 = g(x_2)$$

By substituting the values for x_1 and x_2 we obtain the following:

$$x_1 = f(x_0); \quad x_2 = u(f(x_0)); \quad x_3 = g(u(f(x_0)))$$

We can remove the assignments for x_1 and x_2 as they are now redundant. So now we have $x_3 = g(u(f(x_0))$ which corresponds to the statement $x = g(u(f(x)))$.

This conversion from assignments to simultaneous equations is similar to converting code to SSA (Static Single Assignment) form which is often used in conjunction with compiler optimisations (for example, [33] gives details about how to compute SSA form). In SSA form, each definition of a variable is given a different name and each use is renamed

according to the appropriate definition. In the SSA form, when there are different control flow paths, a special statement called a ϕ (phi) function is added. However, as we are only aiming to simplify a set of simultaneous equations, we will not use the SSA form directly. In particular, our proofs will not need to use phi functions as we will use the results of Section 5.1.3 to enable us to deal with **if** and **while** separately and we can obfuscate a sequence of statements by obfuscating the individual statements (it will suffice for our purpose to prove individual obfuscated statements correct since by Equation 5.13, proving correctness of a block of obfuscated statements is equivalent to composing the proofs of correctness of each statement in the block) . We will only use the SSA form as a guide to help us to specify a set of simultaneous equations which we can manipulate and simplify.

5.1.5 Steps in a proof

There are four main steps in constructing our proofs of correctness.

Simultaneous Equations The first step is to convert a sequential program into a set of simultaneous equations using the SSA form as a guide. This means that each new definition of a variable has a unique subscript and each use of a variable should refer to the previous instance of the variable.

Substitution Once we have converted our sequential code to a set of simultaneous equations then the next phase is to reduce the set of equations by performing substitutions. However, sometimes problems can arise.

Suppose that we have the following set of simultaneous equations:

$$y_1 = x_0 + 1; \quad x_1 = x_0 + 2; \quad y_2 = y_1 - 1$$

Substituting the value for y_1 gives

$$y_1 = x_0 + 1; \quad x_1 = x_0 + 2; \quad y_2 = x_0$$

We can see that the "last" definition for x is at x_1 but the expression for y_2 uses an earlier definition of x. Whenever this type of situation occurs then we cannot immediately convert such sets of equations back to sequential code. The last step discusses possible solutions for this problem.

Redundant Definitions The next step after substitution is to remove redundant definitions. A definition $x_i = e$ is *redundant* in a set of simultaneous equations if no equation uses x_i and there exists some definition $x_j = e'$ where $j > i$. This latter condition ensures that we do not remove the "last" definition of a variable (and since we convert using a form of SSA we know that the last definition of a variable will have the largest subscript).

Converting back Once the set of simultaneous equations has been reduced they need to be converted to sequential code. As mentioned earlier, sometimes we cannot immediately convert the set of equations back to sequential code. For example, suppose that after substitution and refinement we are left with the following pair of simultaneous equations:

$$x_1 = x_0 + 2; \quad y_2 = x_0$$

This cannot be converted to:

$$x = x + 2; \quad y = x$$

as the final value of y in this sequence is equivalent to x_1 not x_0 as required. One solution is to introduce a new variable which holds the value of x_0:

$$t_1 = x_0; \quad x_1 = x_0 + 2; \quad y_2 = t_1$$

So this can be converted to:

$$t = x; \quad x = x + 2; \quad y = t$$

As an alternative, we could convert the simultaneous equations to:

$$x = x + 2; \quad y = x - 2;$$

However this kind of rewriting is not always possible in general.

5.2 Proving obfuscations correct

In this section we prove the correctness of slicing obfuscations described in the previous chapter using the refinement framework described above.

5.2.1 Correctness of a bogus predicate

In 4.4.2, a bogus predicate of the following form was used:

$$x = G;\ S; \quad \rightsquigarrow \quad x = G;\ \textbf{if}\ (q^T)\ S;\ \textbf{else}\ y = H(x); \tag{5.14}$$

where S is a statement and q^T is a true predicate.

To prove that this is a correct transformation, we can go back to our statement models from Section 5.1.1. We can see that with an initial state of σ_0 and an initial value x_0 for x the final state for both blocks in Equation (5.14) is:

$$S(\sigma_0 \oplus \{x \mapsto G[x_0/x]\})$$

5.2.2 Correctness of a variable encoding

In 4.4.3, a variable encoding of the form

$$x \rightsquigarrow \alpha * x + \beta * y$$

where α and β are constants, was discussed.

For an obfuscation which transforms a variable x to $\alpha * x + \beta * y$, we have three rewrite rules:

- (use of x) $U(x) \rightsquigarrow U(\frac{x-\beta*y}{\alpha})$
- (def of x) $x = E \rightsquigarrow x = \alpha * E' + \beta * y$ where $E' = E[\frac{x-\beta*y}{\alpha}/x]$
- (def of y) $y = f(x) \rightsquigarrow \left\{ \begin{array}{l} t = x - \beta * y; \\ y = f(t/\alpha); \\ x = t + \beta * y; \end{array} \right\}$

Figure 5.2: Rewrite rules for a variable encoding

For this type of data transformation, the abstraction function is

$$af \equiv x = (x - \beta * y)/\alpha$$

and the conversion function is

$$cf \equiv x = \alpha * x + \beta * y$$

It is worth nothing that this obfuscation is not suitable if multiplication can overflow and division is not exact.

For this transformation we have three rewrite rules which are summarised in Figure 5.2. First, we transform a use of x, say $U(x)$, to $U(\frac{x-\beta*y}{\alpha})$. Next, using Equation (5.10) an assignment to x is transformed as follows:

$$x = E \rightsquigarrow x = \alpha * E' + \beta * y \text{ where } E' = E[\frac{x - \beta * y}{\alpha}/x]$$

For example

$$\begin{array}{rcl} x{+}{+} & \rightsquigarrow & x = \alpha * ((x - \beta * y)/\alpha) + 1) + \beta * y \\ & \equiv & x = x - \beta * y + \alpha + \beta * y \\ & \equiv & x = x + \alpha \end{array}$$

Proof. □

$$
\begin{array}{ll}
& cf; \mathcal{O}(B); af \\
\equiv & \{\text{definitions}\} \\
& x = \alpha * x + \beta * y; \\
& t = x - \beta * y; \\
& y = f(t/\alpha); \\
& x = t + \beta * y; \\
& x = (x - \beta * y)/\alpha; \\
\equiv & \{\text{sim eqns}\} \\
& x_1 = \alpha * x_0 + \beta * y_0; \\
& t_1 = x_1 - \beta * y_0; \\
& y_1 = f(t_1/\alpha); \\
& x_2 = t_1 + \beta * y_1; \\
& x_3 = (x_2 - \beta * y_1)/\alpha;
\end{array}
\qquad
\begin{array}{ll}
\equiv & \{\text{sub } x_1, t_1 \text{ and } x_2\} \\
& x_1 = \alpha * x_0 + \beta * y_0; \\
& t_1 = \alpha * x_0; \\
& y_1 = f(x_0); \\
& x_2 = \alpha * x_0 + \beta * y_1; \\
& x_3 = x_0; \\
\equiv & \{\text{remove } x_1, t_1 \text{ and } x_2\} \\
& y_1 = f(x_0); \quad x_3 = x_0; \\
\equiv & \{\text{sequential code}\} \\
& y = f(x); \quad x = x; \\
\equiv & \{x = x \equiv skip\} \\
& y = f(x); \\
\equiv & \{\text{definition}\} \\
& B
\end{array}
$$

Figure 5.3: Proof of correctness for a variable encoding showing $cf; \mathcal{O}(B); af \equiv B$

Finally, our obfuscated value for x depends on y; thus whenever we have a definition of y then we will also need a definition of x as well. Suppose that we want to transform the statement $B \equiv y = f(x)$ where f is a function (which could depend on other variables including y). We propose that a suitable transformation is

$$\mathcal{O}(B) \equiv \left\{ \begin{array}{l} t = x - \beta * y; \\ y = f(t/\alpha); \\ x = t + \beta * y; \end{array} \right\}$$

where t is a fresh variable. In Figure 5.3, Equation (5.9) is used to show that:

$$cf; \mathcal{O}(B); af \equiv B$$

5.2.3 Correctness of a **while** loop

In 4.4.4, a new guard variable j was added to introduce dependencies amongst the three output variables in the Word Count example. To add guards in **while** loop conditionals,

we have the following two choices:

$$\textbf{while } (c)\ S \rightsquigarrow \textbf{while } (c\ \wedge p)\ S$$
$$\textbf{while } (c)\ S \rightsquigarrow \textbf{while } (c\ \vee q)\ S$$

We now show under what conditions these transformations are valid.

Let us consider the conditions for adding the predicate q as a disjunction. When c is $true$ then $c \vee q$ is also $true$ but when c is $false$ we want $c \vee q$ to be $false$ as well, *i.e.*

$$\begin{aligned}
& \neg c \Rightarrow \neg(c \vee q) \\
\equiv \quad & \{\text{de Morgan's Law}\} \\
& \neg c \Rightarrow (\neg c \wedge \neg q) \\
\equiv \quad & \{\text{distribution of } \Rightarrow\} \\
& (\neg c \Rightarrow \neg c) \wedge (\neg c \Rightarrow \neg q) \\
\equiv \quad & \{\text{idempotence of } \Rightarrow\} \\
& true \wedge (\neg c \Rightarrow \neg q) \\
\equiv \quad & \{\text{unit of } \wedge\} \\
& \neg c \Rightarrow \neg q \\
\equiv \quad & \{\text{contrapositive}\} \\
& q \Rightarrow c
\end{aligned}$$

If we add p as a conjunction then when c is $false$ then $c \wedge p$ will also be $false$. When

c is *true* then we also want $c \wedge p$ to be *true*, *i.e.*

$$
\begin{array}{ll}
 & c \Rightarrow c \wedge p \\
\equiv & \quad \{\text{distribution of } \Rightarrow\} \\
 & (c \Rightarrow c) \wedge (c \Rightarrow p) \\
\equiv & \quad \{\text{idempotence of } \Rightarrow\} \\
 & \mathit{true} \wedge (c \Rightarrow p) \\
\equiv & \quad \{\text{unit of } \wedge\} \\
 & c \Rightarrow p
\end{array}
$$

Adding a guard to a **while** loop predicate is a simple but effective obfuscation. But for expressing the power of this framework, we illustrate the correctness of another **while** loop obfuscation.

In 4.1.3, a Sum Product example was discussed. For notational convenience we, omit typing information and provide the following code snippet:

$$
\begin{array}{rl}
 & i = 1;\ x = 0;\ y = 1; \\
P \ \equiv & \textbf{while}\ (i < n)\{ \\
 & \quad i = i + 1;\ x = x + i;\ y = y * i;\} \\
 & \textbf{out}(x);\ \textbf{out}(y);
\end{array}
$$

Since, i is a loop variable, we can use the mapping $i \rightsquigarrow 2 * i$ to give the following obfuscated program:

$$
\begin{array}{rl}
 & i = 2;\ x = 0;\ y = 1; \\
\mathcal{O}(P) \ \equiv & \textbf{while}\ (i < 2 * n)\{ \\
 & \quad i = i + 2;\ x = x + (i/2);\ y = y * (i/2);\} \\
 & \textbf{out}(x);\ \textbf{out}(y);
\end{array}
$$

$af; P$
$\equiv$ {definitions}
$af;\ i = 1;\ x = 0;\ y = 1;$
while $(i < n)\{$
$\quad i = i + 1;\ x = x + i;\ y = y * i;\}$
out$(x);$ **out**$(y);$
$\equiv$ $\{cf; af \equiv skip\}$
$af;\ i = 1;\ x = 0;\ y = 1;\ cf;\ af;$
while $(i < n)\{$
$\quad i = i + 1;\ x = x + i;\ y = y * i;\}$
out$(x);$ **out**$(y);$
$\equiv$ {Equation (5.3)}
$i = 2;\ x = 0;\ y = 1;\ af;$
while $(i < n)\{$
$\quad i = i + 1;\ x = x + i;\ y = y * i;\}$
out$(x);$ **out**$(y);$
$\equiv$ {Equations (5.12) and (5.8)}
$i = 2;\ x = 0;\ y = 1;$
while $((i/2) < n)\{$
$\quad af;\ i = i + 1;\ x = x + i;\ y = y * i;\ cf\};\ af$
out$(x);$ **out**$(y);$
$\equiv$ {Equation (5.3)}
$i = 2;\ x = 0;\ y = 1;$
while $((i/2) < n)\{$
$\quad i = 2 * ((i/2) + 1;\ x = x + (i/2);\ y = y * (i/2);\};\ af$
out$(x);$ **out**$(y);$
$\equiv$ {exact arithmetic}
$i = 2;\ x = 0;\ y = 1;$
while $(i < 2 * n)\{$
$\quad i = i + 2;\ x = x + (i/2);\ y = y * (i/2);\};\ af$
out$(x);$ **out**$(y);$
$\equiv$ {definitions}
$\mathcal{O}(P);\ af$

Figure 5.4: Proof of correctness for a **while** loop

The refinement functions for this obfuscation are:

$$cf \equiv i = 2 * i \quad af \equiv i = i/2$$

To prove that $\mathcal{O}(P)$ is correct we use Equation (5.8) to show that:

$$af;\ P \ \equiv \ \mathcal{O}(P);\ af$$

This proof is given in Figure 5.4.

5.2.4 Correctness of a variable split

Suppose that we split an integer variable v into two variables a and b. Using [36], we can write the relationship between v and the split components as

$$v \rightsquigarrow \langle a, b \rangle$$

We need three functions g (the abstraction function), f_1 and f_2 (the conversion functions) such that

$$a = f_1(v) \qquad b = f_2(v) \qquad v = g(a, b)$$

which satisfy $v = g(f_1(v), f_2(v))$ and is subject to the invariant $I(a, b)$.

A use of v, $U(v)$, is replaced by $U(g(a, b))$. An assignment to v needs to be replaced by two assignments:

$$\begin{aligned} v = E \rightsquigarrow \{a = f_1(E');\ b = f_2(E')\} \\ \text{where } E' = E[g(a, b)/v] \end{aligned} \tag{5.15}$$

These transformations need to be applied exhaustively to the whole method (or at the very least to the whole scope of v).

In 4.4.5, the $\mathcal{O}(S)$ was:

$$\textbf{if } (b == 9) \textbf{ then } \{a = a + 1;\ b = 0\} \textbf{ else } \{b = b + 1\}$$

In Figure 5.5 we show this is correct by using Equation (5.9) to prove that $S \equiv cf; \mathcal{O}(S); af$.

5.2.5 Correctness of an array transformation

In 4.4.6, we performed a simple array transformation that converted the variables nl, nw, and nc to:

$$nl \rightsquigarrow a[0] \qquad nw \rightsquigarrow a[1] \qquad nc \rightsquigarrow a[2]$$

We noted that these this transformation was a simple variable renaming and was enough to decrease the residue size (since arrays are treated as an aggregate data type by a slicer and no distinction between the indices are made). Since variable renaming produces no state transformation, the proof framework is not required to prove the correctness of our simple array transformation (intuitively it can be seen that each of the original three variables are mapped to successive array indices and their use is also in the same order). Thus, complicated array transformations such as splitting, merging, folding, and flattening

Proof.

$$
\begin{array}{ll}
 & cf; \mathcal{O}(S); af \\
\equiv & \quad \{af; cf \equiv skip\} \\
 & cf;\ \textsf{if}\ (b == 9) \\
 & \quad \textsf{then}\ \{af;\ cf;\ a = a + 1;\ b = 0;\ af;\ cf\} \\
 & \quad \textsf{else}\ \{af;\ cf;\ b = b + 1;\ af;\ cf\};\ af \\
\equiv & \quad \{\text{Equation (5.11)}\} \\
 & cf;\ \mathcal{O}(\textsf{if}\ ((nc\,\%\,10) == 9) \\
 & \quad \textsf{then}\ \{cf;\ a = a + 1;\ b = 0;\ af\} \\
 & \quad \textsf{else}\ \{cf;\ b = b + 1;\ af\});\ af \\
\equiv & \quad \{\text{definitions and Equation (5.9)}\} \\
 & \textsf{if}\ ((nc\,\%\,10) == 9) \\
 & \textsf{then}\ \{a = nc\,/\,10;\ b = nc\,\%\,10;\ a = a + 1; \\
 & \qquad b = 0;\ nc = 10 * a + b\} \\
 & \textsf{else}\ \{a = nc\,/\,10;\ b = nc\,\%\,10;\ b = b + 1; \\
 & \qquad nc = 10 * a + b\} \\
\equiv & \quad \{\text{simultaneous equations in branches}\} \\
 & \textsf{if}\ ((nc_0\,\%\,10) == 9) \\
 & \textsf{then}\ \{a_1 = nc_0\,/\,10;\ b_1 = nc_0\,\%\,10; \\
 & \qquad a_2 = a_1 + 1;\ b_2 = 0;\ nc_1 = 10 * a_2 + b_2\} \\
 & \textsf{else}\ \{a_3 = nc_0\,/\,10;\ b_3 = nc_0\,\%\,10; \\
 & \qquad b_4 = b_3 + 1;\ nc_2 = 10 * a_3 + b_4\} \\
\equiv & \quad \{\text{substitutions}\} \\
 & \textsf{if}\ ((nc_0\,\%\,10) == 9) \\
 & \textsf{then}\ \{nc_1 = 10 * (nc_0\,/\,10) + 10\} \\
 & \textsf{else}\ \{nc_2 = 10 * (nc_0\,/\,10) + (nc_0\,\%\,10) + 1\} \\
\equiv & \quad \{\text{modular arithmetic}\} \\
 & \textsf{if}\ ((nc_0\,\%\,10) == 9)\ \textsf{then}\ \{nc_1 = nc_0 + 1\} \\
 & \qquad\qquad\qquad\qquad \textsf{else}\ \{nc_2 = nc_0 + 1\} \\
\equiv & \quad \{\text{convert back to assignments}\} \\
 & \textsf{if}\ ((nc\,\%\,10) == 9)\ \textsf{then}\ \{nc = nc + 1\} \\
 & \qquad\qquad\qquad\qquad \textsf{else}\ \{nc = nc + 1\} \\
\equiv & \quad \{\text{identical branches}\} \\
 & nc = nc + 1
\end{array}
$$

□

Figure 5.5: Proof of correctness for the transformation using a variable split

(outlined in Chapter 2) were not deemed necessary for obtaining a smaller residue in our particular attack model. An interested reader is referred to [37] for a detailed discussion on how array transformations could be proved correct in the functional programming domain.

5.3 Applying the transformations

In this section we discuss some of the available choices to make while applying our obfuscations.

5.3.1 Placing the transforms

When determining where to place our obfuscating transforms we used backwards slices of the program to guide us — in particular, we concentrated on orphaned points in the residues of the output variables. For transformations such as encodings and variable splitting, we need to apply them to, at least, to the scope of the variable. With other transformations, such as placing bogus predicates, we can make a choice in placing our obfuscations.

When slicing for a particular variable, say y, we have an assignment $x = G$ in the residue of y then we can add a dependency for y by using the transformation in Equation (5.14). If we have a number of assignments for x then, as we consider backwards slices, we pick the last assignment for x.

Suppose we have the following code fragment:

$$x = E;\ S;\ x = F;$$

where S is a block of statements in which x is used but not defined and F is an expression which does not depend (directly or indirectly) on x. This means that the assignment $x = F$ kills the previous definition of x and the backwards slice for x may not contain the earlier definition for x. We can perform the following example transformation:

$$x = F; \quad \leadsto \quad \textbf{if}\ (p^T)\ x = F;\ \textbf{else}\ x{+}{+};$$

Now the backwards slice for x should include the previous assignment to x.

Many of our transforms relied on the use of bogus predicates, which were a simplified form of *opaque predicates* (see Chapter 2). The predicates in our examples used invariants

that we, as the creator of a method, knew to be true. When deciding where to place a bogus predicate, we should determine what invariants we could use and pick a place that has an invariant which seems hard for an attacker to determine. Similarly, loop transformation in Section 4.4.4 was effective in reducing the sizes of the residues with only a small increase in the method size — but this transformation is only applicable if our method naturally contains a loop. If we can "fake" a **while** loop then we can easily apply the loop transformations. Suppose that we have a block of code B and the state before B is σ. Then we need to find a predicate p such that $p(\sigma)$ is true but $p(B(\sigma))$ is false. Armed with such a predicate we can perform the following transformation:

$$B \rightsquigarrow \textbf{while } (p) \ \{B\}$$

We can then apply our loop transformations.

5.3.2 Localising the transformations

The variable obfuscations proposed in [27] and [38] are applied to an entire program or at the very least the entire scope of a variable. If we apply a data obfuscation to the whole program then we need to convert any input to an obfuscated variable using a conversion function. Any outputs of obfuscated variables need to have the appropriate abstraction function applied to them.

When using conversion functions we can localise an obfuscation to a particular code block. Suppose we have an obfuscation (with functions cf and af) of a variable x and a piece of code with three blocks $A; B; C$ which all define or use x. Then we can obfuscate B separately to obtain $\mathcal{O}(B)$ and so our code becomes:

$$A;\ cf;\ \mathcal{O}(B);\ af;\ C$$

Note that since $cf; \mathcal{O}(B); af \equiv B$ then we must ensure that we do not "reduce" this code sequence otherwise we will "deobfuscate" $\mathcal{O}(B)$.

If we had three different data obfuscations (say $\mathcal{O}_A$, $\mathcal{O}_B$ and $\mathcal{O}_C$ with appropriate conversion functions) then we can obfuscate the blocks A, B and C separately and sequentially compose the results:

$$cf_A;\ \mathcal{O}_A(A);\ af_A;\ cf_B;\ \mathcal{O}_B(B);\ af_B;\ cf_C;\ \mathcal{O}_C(C);\ af_C$$

This means that we can have regions in the program in which we can apply different obfuscations to the same variable and so we can create a "scope" for an obfuscation. To help disguise the conversions we should try to combine the expressions for $af_A;\ cf_B$ and $af_B;\ cf_C$.

5.3.3 Combining transformations

Since we are considering our obfuscations as functions we may naturally want to compose obfuscations. For some variable x, suppose that we have two obfuscations $\mathcal{O}_1$ and $\mathcal{O}_2$. For these obfuscations, the conversion functions are $cf_1 \equiv x = f_1(x)$ and $cf_2 \equiv x = f_2(x)$ (with corresponding abstraction functions $af_1 \equiv x = g_1(x)$ and $af_2 \equiv x = g_2(x)$). To obfuscate a statement S by applying $\mathcal{O}_1$ followed by $\mathcal{O}_2$ we have:

$$\mathcal{O}_2(\mathcal{O}_1(S)) \equiv af_2;\ af_1;\ S;\ cf_1;\ cf_2$$

This is equivalent to having a single obfuscation $\mathcal{O}_{1;2}$ with conversion function $cf_{1;2} \equiv x = (f_2 \cdot f_1)(x)$ and abstraction function $af_{1;2} \equiv x = (g_1 \cdot g_2)(x)$. We define $\mathcal{O}_{1;2} \equiv \mathcal{O}_2 \cdot \mathcal{O}_1$.

For example, we can apply a variable transformation to array elements. So using the function $\lambda x.(2 * x + 1)$ we could have the following array conversion between the arrays A and B:

$$cf \equiv B[i] = 2 * A[i] + 1$$

Since i acts as a dummy variable we can write transformations which depend on i:

$$cf \equiv B[i] = A[i] + i$$

We can combine a variable transformation with array obfuscations given in [39]. For instance if we had the functions $f :: \mathbb{Z} \to \mathbb{Z}$ and $p :: [0..n) \to [0..n)$ (with appropriate inverses) then here is a possible conversion function

$$cf \equiv A[i] = f(A[p(i)])$$

in which f acts as a variable transformation and p is an array index permutation.

Another way to combine obfuscations is to overlap their scope. For instance suppose we have the following blocks of code: A; B; C and we have two data obfuscations $\mathcal{O}_1$ applied to $A; B$ and $\mathcal{O}_2$ applied to $B; C$. Then we have:

$$\mathcal{O}_1(A);\ \mathcal{O}_{1;2}(B);\ \mathcal{O}_2(C)$$

A simple example in Figure 5.6 illustrates that the order of applying two obfuscations, namely bogus predicates and variable encoding, does not affect the correctness properties of a block of code.

$x = G(x);$

$\equiv$ {bogus predicate}
$x = G(x);$
if $p^F(x)$
$y = H(x);$

$\equiv$ {variable encoding}
$x = G(x - y) + y;$
if $p^F(x - y)$
$t = x - y;\ y = H(t);\ x = t + y;$

$\equiv$

$x = G(x);$

$\equiv$ {variable encoding}
$x = G(x - y) + y;$

$\equiv$ {bogus predicate}
$x = G(x - y) + y;$
if $p^F(x - y)$
$y = H(x);$

{equivalent since $p^F(x - y)$ is false}

Figure 5.6: An example of composition of obfuscations

For our examples we considered up to three output variables and sometimes it was necessary to add more than one obfuscation to decrease the number of residues. An example is in the case of *encode*3 (from Section 4.4.3) where we used three different encodings (one for each output variable). Further work is needed to study the effects of

composing and ordering obfuscations on residue metrics.

5.4 Conclusion

In [36], Drape provided a framework for proving obfuscations correct in the functional programming domain and provided an extension of using the framework for proving correctness for imperative constructs. The main contribution of this chapter has been to use the extended framework to prove correctness of the slicing obfuscations proposed in the previous chapter. It also discussed several issues to consider while applying the obfuscations such as their placement, localisation, and combination.

A couple of interesting future research directions result from the techniques outlined in this chapter. The first one is to do with the use phi functions in SSA form. One obvious advantage of using phi functions would be in the reduction of effort in proving the correctness of each individual statement in a block of obfuscated code. The second interesting research direction is in the area of studying the effect of composing and ordering obfuscating transforms on the residue metrics. For this, we suspect that a study initiated in [52] would be a good starting point since it discusses the use of a weighted finite state automata to achieve "maximum" obfuscation for a given program. The proposed use of weights in the arcs were the software quality metrics such as those by McCabe [88] and Harrison [51]. It would be interesting to see if the model in [52] holds if the weights are replaced by the residue metrics.

6

Designing Distributed Opaque Predicates

Having introduced obfuscations to deter slicing attacks in sequential programs, we now show how to design obfuscations in processes executing in a distributed computing environment. This chapter focuses on obfuscations that are designed to thwart the malicious host problem of mobile agents, which are instances of processes (with added functionalities) executing in distributed computing environments. The novel contribution of this chapter rests in the demonstration of the use of operating characteristics of processes executing in distributed computing environment to design obfuscating transforms. Discussion of obfuscatory strength of these transforms are discussed in the next chapter.

The rest of the chapter is structured as follows: In the next section, we describe the malicious host problem in the context of mobile agents and make a case for obfuscations in form of inter-process dependencies to be used in protecting the applications. We then describe the execution environment for which our obfuscating transforms are designed. This is followed by a description of distributed opaque predicates which are transforms for introducing dependencies between processes (agents) executing in a distributed computing scenario. We also discuss the engineering issues that need to be addressed for constructing these predicates. In the last section, we comment on the obfuscatory strength of the proposed technique by arguing that a class of static and dynamic analysis attacks are infeasible for a range of adversaries.

6.1 The malicious host problem of mobile agents

The term "agent" originally evolved in the Artificial Intelligence (AI) community, where an agent was defined as an integrated system performing some tasks on behalf of a user [76]. The recent trend in Distributed Artificial Intelligence (DAI) extends the concept of agents to "software (SW) agents" by considering them as software units capable of being customised and composed to form a complex system [121]. An agent, like an object, has an internal state which reflects its knowledge. The internal state is specified and refined during its lifetime. An agent also has reasoning capabilities that determine its internal behaviour. It is assumed that an agent behaviour may change dynamically at run-time and can be specified in a declarative way. Its external behaviours consist of communicative acts with other agents or control actions on software or hardware devices. An agent system or a multi-agent system (MAS) refers to an agent-based system, which usually consists of numerous agents of different types. The agents in the system should coordinate, cooperate, and sometimes compete among themselves in order to accomplish a given task. A mobile agent is not bound to the system on which it starts execution. It is free to move around among the hosts in its environment. Created on one physical host (execution environment), a mobile agent can transport its state (typically the attribute values of the

agent that help it determine what to do when it resumes execution at its destination) and code (class code necessary for an agent to execute) with it to another execution environment in the network, where it resumes execution. Mobile agents communicate amongst themselves and other hosts with message-passing primitives. The asynchrony of message-passing coupled with the lack of global clock and induced latency lends a distributed computing flavour to the execution environment.

Mobile agents have distinctive advantages over the traditional client-server architecture which make them suitable for a wide array of electronic commerce applications. Claessens *et al.* [26] argue that it is becoming increasingly time consuming and difficult for modern day applications to rely on client-server architecture for their processing. First, with the vast array of information to process (consulting and comparing different sites before performing a transaction), it is often prudent to move the code closer to the source of data rather than transmitting the data over network links. Secondly, mobile devices have slower and intermittent connections which makes the mobile agents paradigm an interesting tool for data gathering activities.

However, the practical deployment of mobile agents in killer applications is thwarted by the "additional" security requirements they warrant (apart from the standard network security requirements of protecting agent code in transit and securing communication channel between one host and another). There are three dimensions to these "additional" security requirements and we briefly describe them below [26]:

Malicious Users In the context of mobile agents, malicious users can pose a threat to the normal functioning of the system. The user can eavesdrop and tamper execution and communication of the agents, deny having sent an agent or in an electronic commerce environment, refused the transactions made by an agent. Access control policies and authorisation frameworks have generally been proposed to tackle the problem of malicious users.

Malicious Agents In unmonitored environments, mobile agents with malicious intentions could exist. Malicious agents may try to steal a host's private keys, for instance

and attack other agents in the system. Several approaches using agent authentication (allowing a host to identify an agent and sign agent code), sandbox execution (under limited privileges), and proof carrying code have been proposed in the literature to deter attacks by malicious agents on the system [95, 114].

Malicious Hosts Here, agents executing on untrusted platforms/hosts are vulnerable to attacks and thus may leak out trade secrets they contain. Hohl in [56] describes the attack scenario with respect to the Mole mobile agents platform where an agent is completely under the control of a host and the host can: spy out agents' code and data, manipulate agent's code and data, deny execution, and return wrong results to system calls, amongst other things. It is worth noting that the malicious host is a harder problem to solve compared to the malicious agents problem and very few solutions of ad-hoc nature exist to tackle this problem.

We highlight a couple of scenarios where the malicious host problem compromises the security of real life applications. In the first example, consider an agent-oriented electronic fare bidding scenario where agents owned by a travel planning site (such as Orbitz and Expedia) are deployed to execute on hosts owned by travel agents so that they could collect information about the lowest fare that is being offered for a particular itinerary. The agent code may contain privileged information such as reserve price of a fare and variables to store the lowest price offered by each travel agent. One security objective is to keep such information confidential to the travel agents at least for the duration of the transaction and a malicious travel agent host may try to reverse engineer agent code to find out the values of such secret variables. Similarly, consider a grid computing scenario, like the SETI@home project [97], where scientific computation modules are downloaded on untrusted personal computers connected to the global network of loosely-coupled machines. These machines are owned by users willing to contribute a portion of their machine's processing power and time for helping the project compute a section of its scientific result by executing the downloaded code. Here too, it may be desirable that scientific computation logic be kept obscure to the owner of the host so that the owner

finds it hard to find the falsify the execution logic of the downloaded code.

We now discuss the different dimensions of attacks that a malicious host can mount on mobile agents using the taxonomy described by Claessens *et al.* [26].

Protecting Data When mobile agents migrate between several hosts, some potentially malicious, it is desirable that the information they contain be protected from attacks. Several cryptographic solutions in the form of "chain of encapsulated information" have been proposed so that data confidentiality (only the owner of a mobile agent can extract information), non-repudiation (if an offer is made by a host), and forward privacy (identities of previous hosts) are preserved. These solutions are based on available cryptographic protocols where hosts digitally sign the data they give to an agent, and the data are encrypted with the public key of the agent owner.

Execution Integrity This is about ensuring the correct executing of an agent code. First, it needs to be ensured that a host receives correct code and state of an agent. This can be achieved by digitally signing code and state and then encrypting with the public key of the recipient host. Solutions for enforcing execution integrity insist that it becomes possible to check whether a recipient host has executed an agent correctly. "State appraisal", for instance, can express invariants that can be later checked for satisfaction. Through "Cryptographic traces", digitally signed logs of operations performed by an agent can be recorded. "Holographic proofs" are similar to error-correcting codes which check few random bits of agent code for correctness. Execution integrity can also be enforced using "replication" where multiple agents with the same functionality are sent to different hosts for execution and if some of them have been tampered with, voting can find out which of the hosts are malicious.

Execution Privacy Execution privacy is a desirable feature in mobile agents applications where an extra layer of protection, in addition to execution integrity, is required. This property is the most difficult of all three to achieve. "Function hiding" is the most prevalent solution used to ensure execution privacy. It is cryptographic solution which is one-round in nature. Using the illustration from Claessens *et*

al. [26], if the mobile agent owner Alice has a mobile agent which computes a function f and wants to send it to Bob who has data x, she would want the mobile agent to compute $f(x)$ without Bob getting to know f (the fact that Bob may not want to reveal x to Alice is not a requirement here and neither is sending x to Alice an option for mobile agents). Using function hiding, Alice encrypts f, sends $E(f)$ to Bob who then computes $(E(f))(x)$ and sends it back to Alice. Alice then calculates $f(x)$ using the key she used to encrypt f.

The "function hiding" (also called "mobile cryptography") approach for enforcing execution privacy of mobile agents by Sander *et al.* [111] has the drawback that an attacker can learn essential information about the agent by arbitrarily executing it multiple number of times. Hohl [56] pointed out yet another limitation of this method. He observed that mobile cryptography is not applicable to generic agent codes since it has the restriction that agents can send cleartext data to only trusted hosts (for example, the remote entrusting schemes such as [16]).

Code obfuscation was suggested as a solution to enforce execution privacy by Hohl in [56]. He extended the concept of "blackbox security" by incorporating time-limitedness. In this scheme, an agent is considered blackboxed if for a certain known time interval, its code and data cannot be read or modified. However, attacks after the time interval are allowed but assumed to have no effect on the outcome of agent execution. We observe that Hohl's method makes explicit assumption of synchronised global clock for token passing between untrusted servers and is therefore difficult to apply in mobile agent interaction scenario which is inherently distributed in nature. Moreover, his technique fails to correlate between the obfuscation techniques and the corresponding time-limitedness they guaranteed. This problem was also outlined in the "Self-Protecting Mobile Agents" project undertaken by Network Associates Laboratories [34]. Their approach was based on developing tools that would translate an individual agent into a distributed set of tamper-resistant aglets that are never entirely vulnerable to a single host. Three important components of their design were the partitioning of distributed state for an agent

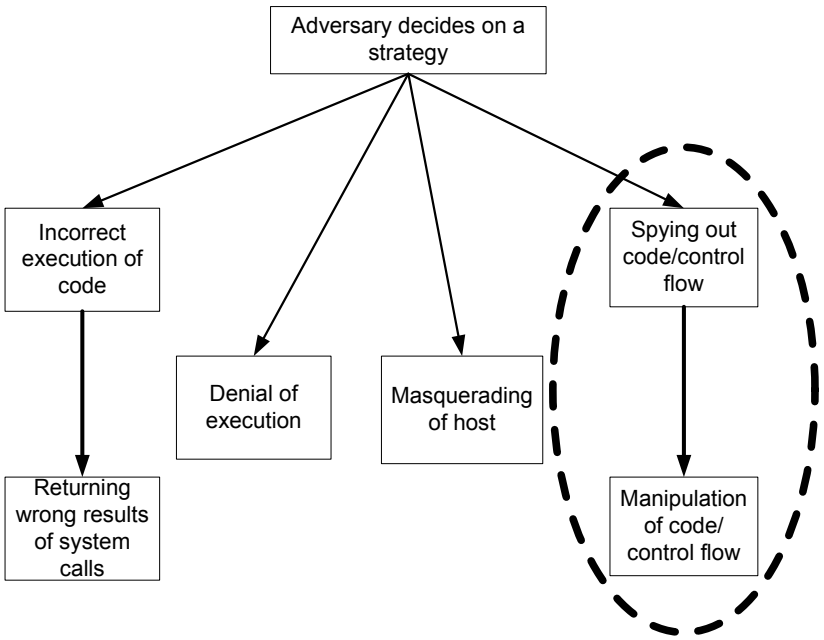

Figure 6.1: The *attack tree*. The class of attacks marked with the dashed oval are specifically addressed in this chapter.

(which is to be protected), applying randomly selected obfuscation algorithms to the aglets, and self-monitoring for tamper detection. The project failed on the grounds that the obfuscations (proposed originally by Collberg et al. [27]) were not related to the distributed agent states. Collberg et al. 's obfuscations were not tuned for adapting to the distributed nature of mobile agents and the authors in [34] observed that obfuscation, in general, is inadequate for providing protection to agents.

In an earlier contribution, we proposed a technique for creating obfuscations using the shared state of agents operating in distributed computing environment [83]. In this simple formulation, a set of obfuscated agents share and pass a data structure in circular fashion and evaluate the outcome of opaque predicates using values of nodes in this data structure. We extend this idea to a greater depth in this chapter and use inherent causality relations between states in distributed computing to influence the outcome of such opaque constructs. The branch confidentiality of Figure 6.1 is achieved by obscuring the real control-flow of behaviours behind irrelevant statements that do not contribute to the actual computations. An adversary with no semantic understanding of correct control-flow of the code will also find it hard to do purposeful manipulation of the code.

It is worth nothing that for the rest of the chapter, we use the terminologies agents and processes interchangeably. As outlined at the start of this section, mobile agents are distributed computing processes with the added features of autonomy and mobility. Since our obfuscations are purely based on the properties of distributed computing, we will often refer to the agents executing in a distributed computing environment as processes.

6.2 Distributed computing and global properties

The objective of this section is to introduce the model of distributed computing on which our obfuscation is based. We will describe the properties of the environment, define global and local states, define ordering properties of states, and lastly elucidate a few state detection problems on which our design is based.

Our distributed computing model is a loosely-coupled message-passing system with no shared memory or global clock. We assume that in our model, no messages are altered or spuriously introduced. Individual processes execute within this framework, do computation and exchange messages. The activity of each process is modelled as execution of a sequence of *events* and these events change the *local state* of the process. An event can be in the form of **send** and **receive** communication primitives. A local state [46] is a set of mappings from variables to values and defined as

Definition 3 (Local state)**.** The local state (σ_i) of a process i is defined by all variables and processor registers of the process.

The execution of a process in a distributed computing model can therefore be viewed as a sequence of local states.

Without a global clock, local states can be ordered only based on the notion of *causality*. The concept of causality for events was proposed by Leslie Lamport. It says that two events are constrained to occur in a certain order only if the occurrence of the first may affect the outcome of the second [72]. For our model, this definition can be applied to local states as well since events change the value of local states. Lamport's causality definition with respect to process states can be expressed as

Definition 4 (Happened-before ($\longrightarrow$))**.** For two states σ_a and σ_b, $\sigma_a \longrightarrow \sigma_b$ if and only if σ_a occurs before σ_b in the same process or the action following σ_a is a send of a message and the action preceding σ_b is a receive of that message. For another state σ_c, if $\sigma_a \longrightarrow \sigma_c$ and $\sigma_c \longrightarrow \sigma_b$, then $\sigma_a \longrightarrow \sigma_b$ holds.

Two states for which the happened-before relation does not hold in either direction are said to be *concurrent*.

Definition 5 (Concurrence ($\|$))**.** For two states σ_a and σ_b,

$$\sigma_a \parallel \sigma_b \models (\sigma_a \not\longrightarrow \sigma_b) \wedge (\sigma_b \not\longrightarrow \sigma_a)$$

We need these definitions to define the concept of *global state* on which our obfuscation is based. But, before defining a global state, we define a *cut*

Definition 6 (Cut)**.** A cut is a set of local states consisting of exactly one state from each process in the system.

Definition 7 (Global state/Consistent cut)**.** A global state (or a consistent cut) is a cut in which all states are pairwise concurrent.

The problem of detecting the global state of a distributed system arises in many practical scenarios of debugging and testing of distributed computing software [46]. Imagine that we need to monitor a cluster computing environment to satisfy a global property such as: "If the total load of the cluster exceeds L, then stop accepting more jobs". Now, monitoring this condition is equivalent to monitoring the local states of the nodes (individual loads) and composing them to find a consistent cut where the sum exceeds L. This problem is non-trivial, and may not have a unique answer, since a global observer with a global clock is absent and the monitoring is done at a local node of the cluster. Similar problems of monitoring global states also arise in electronic commerce; for example, "Book the itinerary if the total cost of the trip (airline, car, and hotel reservation) is between \$1000 and \$1200". Is there a way we can utilise the non-triviality of global state detection problem to design our obfuscations?

6.3 Designing distributed opaque predicates

Having described the attack scenarios and the concept of global states in distributed computing, we now show a way of utilising the global state information to design an opaque predicate construct in distributed computing context. We call this obfuscation a *distributed opaque predicate*.

Definition 8 (Distributed opaque predicate)**.** A *distributed opaque predicate* (Φ_D) is an opaque predicate which depends on local states of multiple processes spread across the distributed system for its evaluation. Using the notation from the last chapter for specifying conditionals, we can write

$$\Phi_D :: state^n \rightarrow \mathbb{B}$$

where $state^n$ denotes a tuple of local state from each of the n processes executing in the distributed system.

We revisit the concept *dynamic* opaque predicates of Palsberg *et al.* [103] here. Palsberg's *dynamic* opaque predicates were constant over a single program run but varied over different program runs. Here we extend their concept by designing distributed opaque predicates to be *temporally unstable*. A temporally unstable distributed opaque predicate can be evaluated at multiple times at different program points $(t_1, t_2, ...)$ during a single program execution such that the values $(v_1, v_2, ...)$ observed to be taken by this predicate are not identical, that is, there exists i, j such that $v_i \neq v_j$.

There are a couple of advantages of making distributed opaque predicates temporally unstable. The first one concerns its reusability; one predicate can be reused multiple times to obfuscate different control flows. The second one relates to its resilience against static analysis attacks. As will be explained later, distributed opaque predicate values (v_i) depend on predetermined embedded message communication pattern between different processes participating in maintaining the opaque predicate. The communication pattern serves as an invariant for maintaining the consistency of local states updates and these in

turn make the predicate go true or false at desired program locations.

Structurally, we design distributed opaque predicates to be relational in nature and of the form:

$$\Phi_D : [(a + b + c + \ldots + n) \ \Re \ K]$$

where $(a, b, c, \ldots, n)$ are integers whose values are set by individual processes (this forms the local state of the process, as explained in the next section), $\Re$ denotes an equality (inequality) operator such as `'='` (`'!='`) and K is a constant. Relational opaque predicates are stealthy in the following sense: an adversary who discovers a relational construct in a program cannot conclude with absolute certainty that it is a distributed opaque predicate since common conditional constructs appearing in programs are often relational in nature. But the most important purpose of making distributed opaque predicates structurally relational lies in the difficulty of detecting this class of predicates in the context of distributed global state monitoring. We will revisit this problem in Chapter 7 on obfuscatory strength evaluation.

In the next section, we will illustrate how distributed opaque predicates can be generated from an instance of a hard combinatorial problem. An obfuscator will ideally embed distributed opaque predicates automatically in a program and insert *send/receive* primitives for generating a predetermined communication invariant. The communication invariant, in turn, will maintain the consistency of local states, that is, the value of each component in the predicate (Φ_D) so that the predicate holds true (Φ_D^T) or false (Φ_D^F) at predetermined control-flow points determined by the obfuscator in advance. We will argue in the obfuscatory strength evaluation section that to an attacker, predicate value at every obfuscated control-flow seems unknown ($\Phi_D^?$).

6.4 Engineering Distributed Opaque Predicates

We discuss here different engineering issues an obfuscator needs to deal with and a step-by-step formulation for generating distributed opaque predicates in the context of distributed

computing obfuscation.

6.4.1 Selecting *guard* processes

Let us assume that a distributed computing system consisting of a set of n inter communicating processes, denoted by $\{P_1, P_2, P_3, ..., P_n\}$, executes on multiple heterogeneous hosts. Assuming that the control-flow of process P_1 is to be obfuscated using distributed opaque predicates, the obfuscator selects or spawns a certain number of *guard* processes to aid in the obfuscation of P_1. Since processes in distributed systems typically collaborate through message exchanges to achieve a particular task, the set of *guards* could be those processes P_1 frequently communicates with. The actual number of *guards* employed in the obfuscation of a single process may depend dynamically on the availability of processes. However, the obfuscator may spawn dummy processes to serve as *guards* if there are not enough processes in the system to do this task. The basic idea is to distribute the local states formed in the construction of distributed opaque predicate in P_1 amongst the *guards* and embed a communication pattern in the form of *send/receive* calls that will update respective local states of processes to previously known values. The local state update rules and communication pattern embedding are described in the following subsections. Our assumption in this illustration is that only process P_1's obfuscated code is under attack by an adversary — this is also the assumption of malicious host problem (refer to Section 6.1) where only one host in a system of collaborating distributed nodes is considered malicious. The guard processes serve to maintain the correct state of the distributed opaque predicate in process P_1 and cannot be compromised by the adversary.

We illustrate the process interaction architecture in Figure 6.2. For our illustration to follow, we have selected two processes, P_2 and P_3, to serve as *guards* for P_1. Local state for each process i is denoted by the variable $p_i.v$. P_1 could, in turn, serve as a *guard* process for helping in obfuscating any other process within the system but we have excluded that possibility for the sake of keeping this illustration simple.

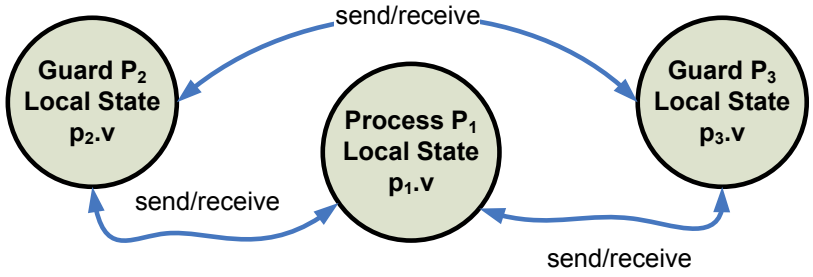

Figure 6.2: The protected process P_1 with local state $p_1.v$ and two *guards* P_2 and P_3 with local states $p_2.v$ and $p_3.v$ respectively.

6.4.2 Incorporating a Knapsack problem instance

We now consider an instance of a hard combinatorial problem called Knapsack problem [45] and show that it can be adapted for manufacturing distributed opaque predicates. The original 0/1-Knapsack problem can be stated as follows:

Definition 9 (Knapsack Problem)**.** Given a set $S = \{a_1, a_2, \ldots, a_n\}$ of positive integers and a sum $T = \sum_{i=1}^{n} x_i a_i$ where each $x_i \in \{0, 1\}$, find the set of x_i.

This decision problem has been shown to be NP-complete [45]. In adapting this problem for manufacturing distributed opaque predicate, the obfuscator selects the set S of positive integers and x_i's according to some predetermined sum T. An adversary through careful static analysis and reverse engineering may come to learn about set S and sum T. However, given an arbitrarily large set, the hard problem for the adversary is to not only decide if a solution vector x exists but to also to determine the vector at precisely the program points $(t_1, t_2, ...)$ where distributed opaque predicates are used to obscure the control-flow of P_1. This is hard since the distributed opaque predicates are constructed from local states (the values range in set S) of *guard* processes and P_1 and local states dynamically change depending on the interaction pattern between processes. This underlying concept will gradually evolve as we describe our methodology.

For our illustration, we select an arbitrary set as:

$$S = \{11, 9, 18, 2, 12, 5, 17, 19, 4, 7, 1, 33\}$$

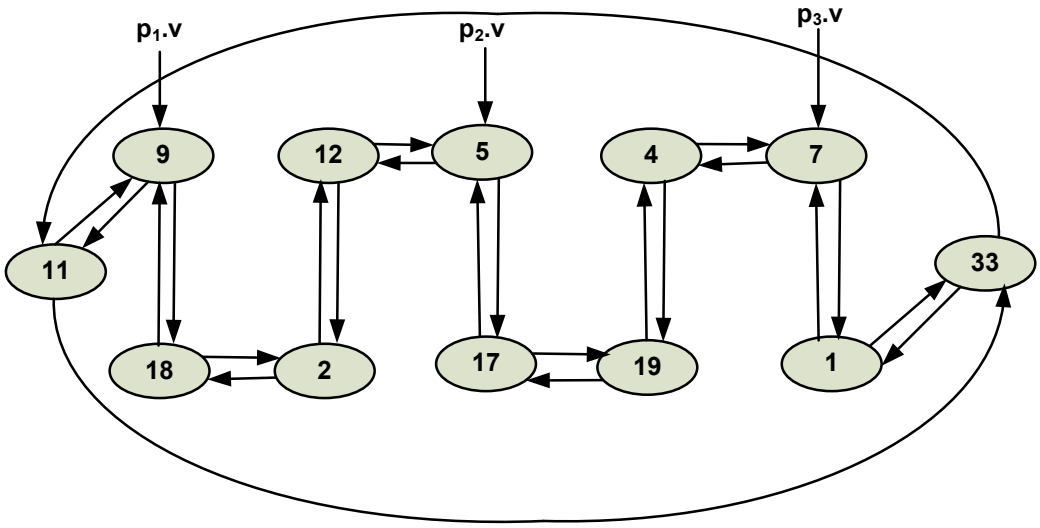

Figure 6.3: The doubly circular linked-list configurations of P_1, P_2 and P_3 initialised with elements from set S. Each copy of the list is also initialised with an initial pointer location ($p_1.v$, $p_2.v$, or $p_3.v$) respective to the process it is sent.

After dynamically selecting/spawning the *guards*, process P_1 and the *guards* are each initialised by passing a dynamic data structure, such as a doubly circular linked-list, initialised with the elements of the set S. This is illustrated in Figure 6.3.

In addition to initialising the linked lists with elements of set S, each copy is also initialised with an initial pointer location respective to the process the list is sent. Node values corresponding to the pointer locations form the local state of that particular process. For our illustration with three processes, the list is initialised with three pointers: $p_1.v$ for P_1, $p_2.v$ for P_2, and $p_3.v$ for process P_3. It is worth noting that $p_i.v$ corresponds to the value of the local state σ_i for process P_i. Messages are exchanged between the *guards* $\{P_2, P_3\}$ and P_1 according to an embedded communication pattern. Generation of this predetermined communication pattern will be discussed shortly. Considering an arbitrary sum for our illustration as $T = 27$, the corresponding solution vector x for the sum T is:

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

For our three process illustration, this solution vector corresponds to $p_1.v = 18$, $p_2.v = 5$, and $p_3.v = 4$. The *distributed opaque predicate* (Φ) thus formed in this case would be:

$$\Phi_D : p_1.v + p_2.v + p_2.v = 27$$

In the following subsections, we will explain how to coordinate the local state update values between processes by controlling their interaction pattern such that Φ_D is satisfied at precisely the program points where the obfuscator decides. We note that there could be other possible solution vectors for set S and a similar approach for constructing distributed opaque predicates could be used with different sets of *guards* and the same sum T. It is worth noting that the opacity of this particular predicate does not rest on the hardness of finding subset x from the given instance of S but rather on the difficulty of program analysis and global state analysis (as explained in details in the Chapter 7). However, the purpose of incorporating a combinatorially hard problem is to add another dimension of difficulty for the attacker — that is, assuming that an obfuscator is successfully able to generate arbitrarily large Knapsack problem instances, the attacker will have to solve these problem instances in addition to performing precise program analysis and global state detection for the successful evaluation of Φ_D.

6.4.3 Defining the local state update rules

Communication in distributed systems is accomplished through the communication primitive events *send(m)* and *receive(m)*, where m denotes the message. In the rest of this chapter, an *event* will refer to either *send(m)* or *receive(m)* unless otherwise specified. In asynchronous message-passing systems, information may flow from one event to another either because the two events belong to the same process, and thus may access the same local state, or because the two events are of different processes and they correspond to the exchange of a message.

These communication primitives can be seen as function on states (analogous to how statements were defined to be functions on states in the last chapter) as follows:

$$\textbf{send} :: state \times message \rightarrow state$$

- **send** :: Present pointer location shifted left of the current node. That is $send(\sigma_a, m) \rightarrow pointer.left(\sigma_a)$ where m denotes the message sent at the local state σ_a of process a.
- **receive** :: Present pointer location shifted right of the current node. That is $receive(m, \sigma_a) \rightarrow pointer.right(\sigma_a)$, where m denotes the message received at the local state σ_a of process a.

Figure 6.4: Local state update rules

and

$$\textbf{receive} :: message \times state \rightarrow state$$

The local state update rules defined on inter-process message communication events are shown in Figure 6.4. Unless explicitly stated, we will omit the notation of local state in the definitions of **send** and **receive**.

Thus, if the local state of P_1, defined by $p_1.v$, is 9 at a certain point in P_1's execution and if P_1 receives a message, the local state will change to 18. Similarly, if P_1 sends a message, the local state changes to 11. Since the distributed opaque predicate is constructed by composing the local states of individual processes and each local state value (corresponding to the pointer location) fluctuates when processes send or receive messages, the predicate will alternate between true/false outcomes throughout the run.

6.4.4 Selection of communication pattern and message types

As stated earlier, local state updates of individual processes take place according to an embedded invariant communication pattern. This predetermined pattern is generated when the embedded *send/receive* calls in processes P_1, P_2, and P_3 get executed. The calls could be embedded by the obfuscator by tracing and annotating the processes with *send/receive* primitives, much in the same way dynamic watermarking algorithms annotate programs for inserting watermark building code [93]. However, there are a couple of problems with adopting this approach for embedding the communication pattern. First of all, embedded *send/receive* calls will generate an arbitrary pattern for each run of the program unless

they are controlled in some way. Secondly, because of the nondeterminism and latency associated with asynchronous message passing, there is no guarantee of causal delivery of messages. Thus, we have to ensure message exchanges satisfy FIFO (First-in-first-out) delivery. This delivery order ensures for all messages m and m' from processes i and j:

$$\textbf{send}_i(m) \longrightarrow \textbf{send}_i(m') \Rightarrow \textbf{deliver}_j(m) \longrightarrow \textbf{deliver}_j(m')$$

Execution of $\textbf{send}_i(m)$ causes message m to be sent from process i. The message is then delivered to the destination j (denoted by $\textbf{deliver}_j(m)$) and subsequently received by executing the **receive** command in process j. This will be further clarified in the Chapter 7.

In distributed systems, the notion of global clock is absent. We propose using vector clocks [87] for solving these two problems. Using vector clock, event orderings based on increasing clock values are guaranteed to be consistent with causal precedence. Before going into a detailed discussion on its usage for constructing the predetermined communication pattern, we briefly provide an overview of vector clocks.

Definition 10 (Vector clock)**.** Vector clock of a system of n processes is an array of n logical clocks, one per process. A local copy of the vector clock is kept in each process P_i, contributing a local state in the construction of distributed opaque predicate. A notation of $VC_i^b[i]$ denotes the logical clock value of P_i at *send/receive* event b. $VC_i^b[j]$ denotes the time $VC_j^a[j]$ of last event a at P_j that is known to have happened before its local event b.

All vector clocks are initialised to be zero (by an initialisation event which precedes every other event when a process starts). The vector clock update rules are given in Figure 6.5.

Three obfuscation-specific message classes are used for message exchanges between process P_1 and the *guards* P_2 and P_3. These message classes help in maintaining consistency of vector clock values and local state updates. Each class is identified by a special tag. The first class is identified by the tag `SYSTEM`. Messages of this class may originate in either P_1, P_2 or P_3 and carry vector timestamp in them. Also, when a process partic-

- If a and b are successive events in P_i, then

$$VC_i^b[i] = VC_i^a[i] + 1$$

- Also, if b denotes *receive(m)* by P_i with a vector timestamp t_m, then

$$VC_i^b[k] = \max(VC_i^a[k], t_m[k])$$

for all $k \neq i$.

Figure 6.5: Vector clock update rules for processes P_i with two successive events a and b.

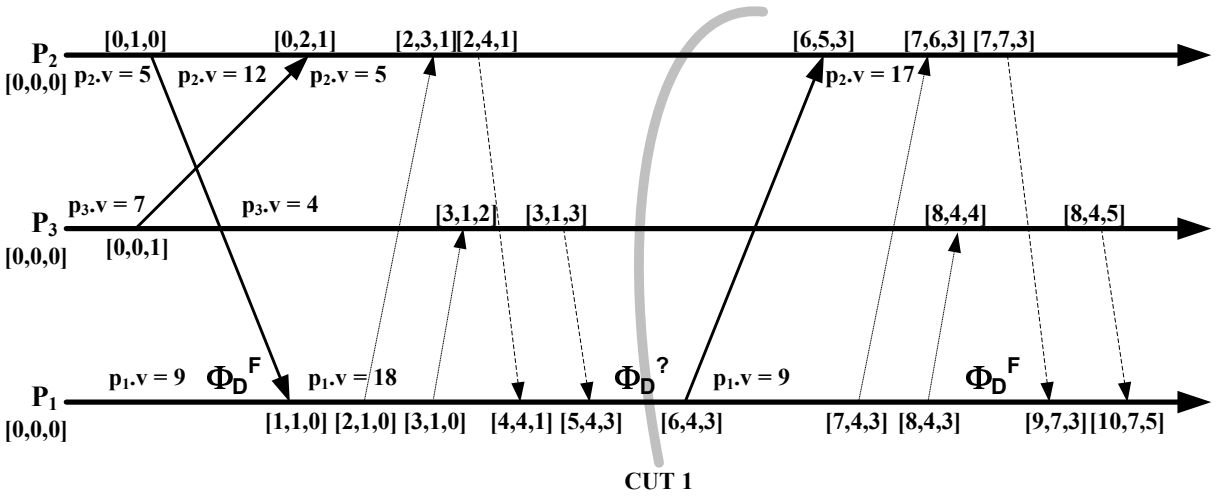

Figure 6.6: The invariant in the form of a predetermined nondeterministic communication pattern is embedded by the obfuscator into P_1, P_2 and P_3. The update pattern of local states can be traced from Figure 6.3. Thick arrows denote `SYSTEM` type messages, thin arrows denote `REQUEST` type messages, and dashed arrows denote `RESPONSE` type messages.

ipates in a *send/receive* event of `SYSTEM` type messages, it updates its vector clock and local state according to the state update rules specified in Section 6.4.3. The second class, `REQUEST`, type message may only originate at P_1 since it is used to request local state values for the *guards*. This class also carries vector timestamp and causes vector clock updates but does not cause any change in local state when received by the *guards*. The third type of message is identified by tag `RESPONSE`. This type is identical to the second class of messages with the exception that these originate at *guards* and are received by P_1. `RESPONSE` messages are used by *guards* to send local state values back to P_1 against incoming `REQUEST` messages.

An example of predetermined communication pattern is illustrated with processes P_1,

P_2, and P_3 in the event-time diagram of Figure 6.6. An event-time diagram maps each event against time and state changes are effected by exchange of messages. The embedded *send/receive* calls in processes P_1, P_2, and P_3 generate this communication pattern. The vector clock value for each participating process is indicated within the square brackets and the value of local state is indicated in variable $p_i.v$, where i denotes the process number. Thick arrows in the figure denote `SYSTEM` type messages. Thin arrows denote `REQUEST` type messages and dashed arrows represent `RESPONSE` type messages. Along the timeline of process P_1, we have labeled the value of predicate (Φ_D) between two successive events distinguished by vector clock values. A (Φ_D^T) label implies that the predicate is guaranteed to hold true within that event interval (successive events) for that particular run. Similarly, (Φ_D^F) implies that the predicate is guaranteed to be false within that interval for that particular run. A label denoted by ($\Phi_D^?$) along the timeline implies that the predicate value is unknown since Φ_D is not guaranteed to hold.

In Figure 6.6, asynchrony of message passing induces nondeterminism within the system since messages can be delivered out of order in the processes. Nondeterminism can cause Φ_D to take on different values depending on the order in which messages are delivered and this would cause difficulty to an adversary trying to find out if Φ_D is guaranteed to hold true or false at a particular point in the event timeline. For instance, in Figure 6.6, it would seem that (Φ) is satisfied at `CUT1` since the local states $p_1.v = 18$, $p_2.v = 5$, $p_3.v = 4$ add up to 27. However, due to nondeterministic nature of message deliveries, it cannot be guaranteed that Φ_D will be satisfied at `CUT1`. In particular, the following two cases could arise out of nondeterminism:

No guarantee on message delivery order

Consider the case from Figure 6.6 where the message (henceforth referred to as message a) originating from P_3 at vector clock value $[0, 0, 1]$ reaches before the message (henceforth referred to as message b) originating at vector clock value $[0, 1, 0]$ of P_2 is sent by process P_2. This situation is depicted in Figure 6.7.

When message a reaches *guard* process P_2, the vector clock value changes to $[0, 1, 1]$

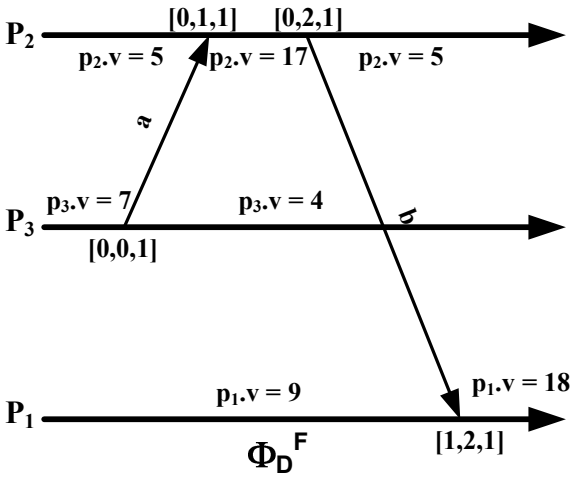

Figure 6.7: No guarantee on message delivery order. Messages a and b are swapped and (Φ_D) is satisfied at CUT1.

and P_2's local state, $p_2.v$, changes from 5 to 17 (refer to Figure 6.3). *Guard* process P_3's local state, $p_3.v$, changes from 7 to 5. However, after P_2 sends message b at vector clock $[0, 2, 1]$, its local state reverts to 5. When the probe messages (REQUEST) are sent by P_1 after the vector clock state $[1, 2, 1]$, the local state values returned from processes P_2 and P_3 are $p_2.v = 5$ and $p_3.v = 4$ respectively. By the time the probe messages are sent to P_2 and P_3, process P_1 has already changed its local state value, $p_1.v$, to 18. The local state values of processes P_1, P_2, and P_3 add up to 27 and the distributed opaque predicate (Φ_D) is satisfied at CUT1.

No guarantee on message delivery

Now consider the case where after the first receive of message b at $[1, 1, 0]$ by process P_1, it cannot be guaranteed that the message a from *guard* process P_3 originating at $[0, 0, 1]$ has reached *guard* process P_2. This guarantee cannot be made because of the nondeterministic nature of asynchronous message-passing. This situation is depicted in Figure 6.8.

As seen from the figure, since *guard* process P_2 changes its local state to $p_2.v = 12$, the local state values of processes P_1, P_2, and P_3 do not add up to 27. Consequently, the distributed opaque predicate (Φ_D) is not satisfied at CUT1. In yet another specialization of this case, message b may reach process P_1 even before message a originates from *guard*

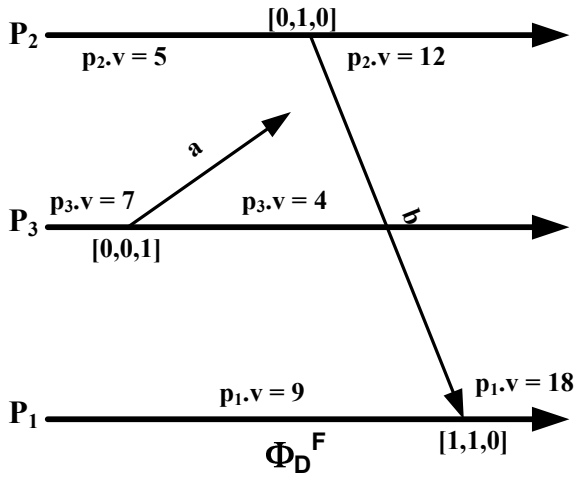

Figure 6.8: No guarantee on message delivery. Message a is in transit while message b reaches process P_1. The predicate (Φ_D) is not satisfied at `CUT1`.

process P_3. In this case, *guard* process P_3 will maintain its local state value at $p_3.v = 7$. Thus, the predicate value will also not be satisfied in this case since the sum of the local state values of processes does not add up to 27.

Thus, we can generally observe, from the asynchronous communication pattern of Figure 6.6, that while designing the communication invariant, crossover message-passing patterns will cause nondeterminism within the system and this property could be utilised by the obfuscator to confuse attackers into falsely believing that a distributed opaque predicate will be guaranteed to hold true or false at a particular program location.

On the other hand, deterministic communication patterns would produce guaranteed results for distributed opaque predicates. An example of deterministic cyclic communication pattern is shown in Figure 6.9. Here, the event-time diagram is a continuation of the one shown in Figure 6.6 and the vector clock ticks are continued along the timelines of processes P_1, P_2, and P_3. The communication pattern is deterministic in the sense that there is only one delivery order for all the messages (unlike in Figure 6.6). Therefore, at vector clock value $[14, 10, 9]$, all previous messages have been delivered and therefore the predicate (Φ_D) is guaranteed to hold true (Φ_D^T) at `CUT2` in process P_1. Similarly, for all other event intervals, (Φ_D) is guaranteed to be false (Φ_D^F). The pattern is cyclic since a message is sent from a process only after the previous expected message has been received

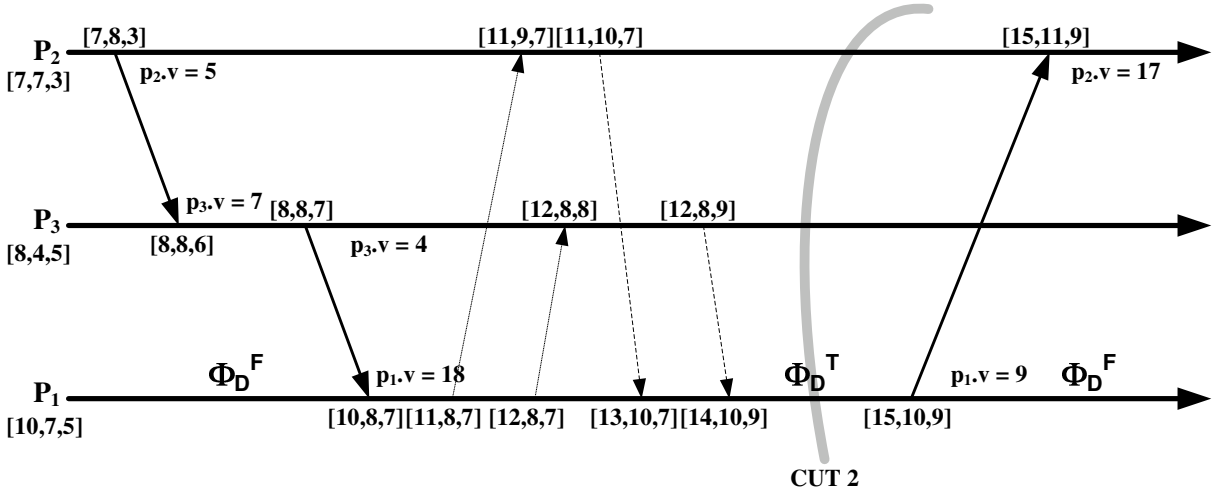

Figure 6.9: The invariant in the form of a predetermined deterministic communication pattern is shown in this figure. As before, thick arrows denote `SYSTEM` type messages, thin arrows denote `REQUEST` type messages, and dashed arrows denote `RESPONSE` type messages.

by it and the pattern both originates and terminates in the guard process P_2.

6.4.5 Embedding distributed opaque predicates

Just as nondeterminism can be used as a technique to generate opaquely unknown predicates, it could also cause problems to the obfuscator since uncontrolled concurrency will update states in an unpredictable way. This is because of the fact that the obfuscator has to be aware at what program points a distributed opaque predicate will hold true or false.

The problem associated with unpredictable local state update can be brought under control if the communication pattern generating code (specifically the **send**/**receive** primitives) can be guarded; i.e., a message contributing to a deterministic communication pattern is only sent from a process if it is guaranteed that the vector clock value of the process issuing this **send** is up-to-date (thereby generating communication patterns such as the one shown in Figure 6.9). Alternatively, this means that the process should have completed all the message communication events (**send**/**receive**) before issuing another **send**. We show an abstract pseudo-code for controlled message passing and predicate evaluation for process P_1 in Figure 6.10. The code follows the conventional programming style of message-passing languages as laid out by MPI ([69])and PVM ([70]).

In Figure 6.10, we have used blocking **receive** to ensure that the local state of process P_1 is consistent before it issues a **send** message (i.e., the process busy-waits on all outstanding messages it has not yet received). For instance, the following snippet

```
while(!receive(VectorClockTimeStamp,SYSTEM,bufferVal)){
probe(ReceivePort); //Check for message by polling
}
```

in process P_1 ensures polls its `ReceivePort` in a continuous loop until a message with tag `SYSTEM` is received in the variable `bifferVal`. The vector clock value is copied in to the variable `VectorClockTimeStamp`. The snippet

```
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp);
```

updates the vector clock of process P_1 following the rules given in Figure 6.5 for a receive message primitive. The function `shift_right()` (`shift_right()`) is responsible for moving the current pointer location on process P_1's copy of the linked list to right (left). The primitive `send(dest,MsgType)` sends a message of type `MsgType` to the destination process `dest`. Before each `send(dest,MsgType)`, the local vector clock is updated by the function `increment(VectorClock)`. To make it more flexible, non-blocking **receive** with guarded *sends* could be used to maintain consistency of local states. This can be implemented by making sure that before each **send** primitive, the vector clock from last **receive** is up-to-date (by comparing it against an expected timestamp value). If the clock is not up-to-date, the process blocks the **send** call for outstanding **receive**s.

The pseudo-code snippets for nondeterministic (`CUT1`) and deterministic (`CUT2`) evaluation of the predicate Φ_D essentially generates the communication patterns of Figures 6.6 and 6.9 respectively. It is worth noting that the codes for both the nondeterministic pattern and the deterministic one look the same but they produce different results for evaluation of Φ_D. At `CUT1`, Φ_D is unknown because the message order in guard process P_2 could be swapped (refer to Figure 6.7) or messages may not be delivered (refer to

Figure 6.8). Thus, for an unknown Φ_D, dummy actions are inserted in branches corresponding to both '`true`' and '`false`' paths. However, at `CUT2`, the obfuscator knows that Φ_D holds true (since all messages sent and received before `CUT2` are guaranteed to be in order). Therefore, the obfuscator inserts real actions in the path corresponding to the '`true`' branch of the control statement. *Guard* process interaction pseudo-code are are similar to P_1's code and have not been included in this illustration.

While obfuscating P_1, the obfuscator may embed many distributed opaque predicates at different control-flow points corresponding to, for example, the construction of watermarking code. Any arbitrary nesting of distributed opaque predicates can be used for obfuscating the control-flows. A different set of *guard* processes could also participate in different communication invariants involving other local state update rules. We recapitulate from Section 6.1 that in the malicious host model of attack, we assume that an adversary can compromise only process P_1. The similarity of communication code for both nondeterministic pattern adds stealth and will make the job of comprehending process P_1's code difficult to an adversary.

6.5 Conclusion

In this chapter, we have addressed the problem privacy protection of mobile agents executing on malicious hosts in distributed computing environments. The main contribution of this chapter is the application of the idea of creating intra-process data dependency from previous chapters to designing inter-process communication dependency and used it to extend the original concept of opaque predicates proposed by Collberg *et al.* to the domain of distributed computing. We also demonstrated that hard combinatorial problems can be tuned with open problems related to distributed systems state monitoring to manufacture a new class of resilient opaque predicates; which we defined in this chapter as distributed opaque predicates.

The work contained in this chapter has been extended from its earlier stages [83, 84, 85]. As with slicing obfuscations, a research challenge is in the automatic embedding and

Process P_1:

```
initialize(VectorClock); //Initialize Vector Clock to [0,0,0]

… //Start nondeterministic predicate evaluation

//get SYSTEM message
while(!receive(VectorClockTimeStamp,SYSTEM,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [1,1,0]
shift_right(p1.v); //point to the right node

//probe for local states from guard processes
increment(VectorClock); // Vector Clock value [2,1,0]
send(P2,REQUEST); //probe for p2.v value
increment(VectorClock); // Vector Clock value [3,1,0]
send(P3,REQUEST); //probe for p3.v value

//get RESPONSE messages
while(!receive(VectorClockTimeStamp,RESPONSE,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
p2.v = bufferVal;
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [4,4,1]
while(!receive(VectorClockTimeStamp,RESPONSE,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
p3.v = bufferVal;
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [5,4,3]

// evaluate distributed opaque predicate
if (p1.v+p2.v+p3.v==27) {// CUT1: Predicate Value Unknown
    // Dummy watermark building code
}
else {
    // Dummy watermark building code
}

… //Start deterministic predicate evaluation

//get SYSTEM message
while(!receive(VectorClockTimeStamp,SYSTEM,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [10,7,8]
shift_right(p1.v); //point to the right node

//probe for local states from guard processes
increment(VectorClock); // Vector Clock value [11,8,7]
send(P2,REQUEST); //probe for p2.v value
increment(VectorClock); // Vector Clock value [12,8,7]
send(P3,REQUEST); //probe for p3.v value

//get RESPONSE messages
while(!receive(VectorClockTimeStamp,RESPONSE,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
p2.v = bufferVal;
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [13,10,7]
while(!receive(VectorClockTimeStamp,RESPONSE,bufferVal)){
    probe(ReceivePort); //Check for message by polling
}
p3.v = bufferVal;
increment(VectorClock);
setMax(VectorClock, VectorClockTimeStamp); // Vector Clock value [14,10,9]

// evaluate distributed opaque predicate
if (p1.v+p2.v+p3.v==27) {   // CUT2: Predicate Value True
    // Real watermark building code
}
else {
    // Dummy watermark building code
}
```

Figure 6.10: Pseudo-code showing the obfuscation of P_1 using distributed opaque predicate Φ_D.

placement of distributed opaque predicates at selected control-flow locations of agents. Also the fault tolerance issues need to be addressed before the idea can be used for real-life applications: What would happen if one or more of the cooperating *guards* accidentally die or are purposefully killed by an adversary? Our present model is rigid in the sense that the loss of *guard* processes will make the obfuscated program go into an incorrect state, thus adding some form of tamper-proofing. But, this notion is weak since the *guards* and messages may be lost in the system accidentally. Further work needs to be done in finding new classes of distributed opaque predicates and instances of hard combinatorial problems for generating them. In Chapter 7 we will address the correctness and evaluation issues of distributed opaque predicates. We will build an attack model and point out the difficulties (from an adversary's perspective) of statically and dynamically observing code which has been obfuscated with distributed opaque predicates.

7

Evaluation of Distributed Opaque Predicates

This chapter's contribution rests in underlying the techniques for evaluating distributed opaque predicates. As a part of this evaluation, we first highlight the correctness properties of distributed programs embedded with distributed opaque predicates using an existing proof technique for message passing programs. Next we evaluate the obfuscatory strength of distributed opaque predicates using Indus, a Java slicer on a set of mobile agents built on JADE platform following the attack process in Section 3.2. We also comment on dynamic analysis attacks that could be mounted on programs obfuscated with distributed opaque predicates.

7.1 Correctness of message-passing programs

In the previous chapter we noted that the resilience of distributed opaque predicates rests on the nondeterminism property of distributed systems having no guarantee on delivery and order of messages. While an obfuscator embedding distributed opaque predicates in a program could carefully control the nondeterminism in the form of guarded **send** and **receive** commands, we stated that it will be difficult for an adversary to reason about the induced nondeterminism from static analysis of the program code. In this section we comment on the correctness properties of message-passing programs obfuscated with distributed opaque predicates.

The nondeterministic behaviour of messages may cause a program to produce different results on the same input data with no modification of the source code [68]. The task of proving correctness of message-passing programs is non-trivial because of the following two reasons; first, we need to be able to use a succinct proof framework that does not rely on enumerating all possible execution paths of a nondeterministic program, and secondly, the existing elaborate proof frameworks using CSP (Communicating Sequential Processes) logic are suitable for proving correctness of only synchronous systems but not for asynchronous ones (our **send** commands do not have matching **receive** commands). Therefore, we resorted to using a "relaxed" CSP framework and notation proposed by Schlichting and Schneider in [113] for proving the partial correctness of message-passing programs. Note that we are stressing on the word "partial" since we are only concentrating on the effects of **send** and **receive** primitives in a message-passing program and assuming the behaviour of the program, otherwise, is correct. Next we outline the proof framework from [113].

7.1.1 Proof framework

In CSP logic, a program P is a collection of n processes $P_1, P_2, \ldots, P_n$, represented by

$$P :: [P_1 \parallel P_2 \parallel \ldots \parallel P_n]$$

These processes execute concurrently and communicate solely by using **input** and **output** commands

$$\textbf{input} : P_1?var$$

and

$$\textbf{output} : P_2!expr$$

The **input** command in process P_2 matches the **output** command in process P_1 if the type of the expression *expr* and the type of the variable *var* are the same. These two commands are always executed synchronously in matching pairs and therefore are blocking in nature. In CSP logic, it is also possible for a command to appear in a guard — a process will wait for the occurrence of the condition in guard. A guard can be represented by expression β which consists of program variables that are accessible to the process in which the guard appears. "Correctness" in a system of distributed processes, ensures that the assertions made in one process are not invalidated by the execution of other processes in the system. These assertions can be on local states of processes before and after it executes a communication primitive (such as send/receive). In other words, correctness property ensures that in a system of asynchronously communicating processes, each process always has its expected local state value after executing a communication command. Schlichting and Schneider observed that the established way of proving correctness using CSP logic involves three steps. First, each process in the system is annotated with assertions, and this forms the *sequential proof*. Next, the assumptions made in the sequential proof about the effects of message receives are validated by doing a *satisfaction proof*. A satisfaction proof ensures that the postconditions of communication commands are true whenever those communication commands terminate. Thus, sequential proofs of processes need assumptions about the effect of communication from satisfaction proofs and satisfaction proofs show these assumptions to be valid by placing constraints on other processes. Lastly, a *noninterference proof* is performed to validate that the execution of each process does not interfere with the assertions made in the sequential proof of other processes. Noninterference is established by proving that no assignment or communication command

interferes with any of the assertions parallel to it, where an assertion I is parallel to an atomic action A if A is contained in one process and I is contained in the proof of another. Furthermore, a statement is an atomic action if it contains at most one reference to at most one shared variable.

The **send** and **receive** primitives we used in the previous chapter are asynchronous in nature. A **send** statement of the form

send *expr* **to** *dest*

sends a message with value computed from *expr* to a process named *dest*. The sender process continues while the message is being delivered and received. A **receive** statement has the form

receive m **when** β

where m is a program variable and β is a Boolean guard expression involving m and other program variables. The process invoking the **receive** statement is delayed until a message with a value, say MSG, is delivered to it and β^m_{MSG} is True (the notation β^m_{MSG} means m is substituted by MSG in every occurrence). A message can be only received if it has been sent and delivered. We will adapt the proof rules proposed by Schlichting and Schneider in [113] and illustrate with simple instances of two-process message passing programs.

Often in proofs of correctness, it is required to relate program variables of one process to program variables of another — and thus, auxiliary variables are used. Auxiliary variables in proof system facilitate assertions in different processes to refer to non-disjoint state spaces. Auxiliary variables, unlike local process variables, are not used in the computation but only in proofs. It is worth noting that proofs of noninterference are needed only if auxiliary variables are used[113] since with disjoint state spaces, execution of one process cannot change the state of another process. But for doing noninterference proofs, every assertion in every process must be compared against every command in every process and against every matching communication pair. This takes a considerable effort.

Although it will affect the completeness of our proofs, we will not go into the complication of proving noninterference of programs since our purpose here is to show how an existing proof framework can be adapted to our distributed programs (and not to propose a new framework for correctness). Interested readers are encouraged to see [113] for learning about techniques for proving noninterference of programs.

7.1.2 Proof rules

Before describing the satisfaction proof rule, we outline the different axioms of the system. The axioms are modelled using multisets, which are sets in which an item may appear more than once. We will use to multiset operators, the difference operator $\ominus$ and the union operator $\oplus$. The multiset difference operator $\ominus$ denotes difference of multisets A and B, where $A \ominus B$ contains those elements of A which are not also elements of B. The multiset union $A \oplus B$ between two multisets A and B comprises of all elements of A together with all elements of B.

First, we have to take into account the messages in the system that have been sent but not received. This defines the following nondeterministic axiom:

Definition 11 (Nondeterministic axiom)**.** Let the auxiliary variables δ_P and γ_P denote the *send multiset* and the *receive multiset* respectively associated with a process P. δ_P contains a copy of every message sent to P and γ_P contains a copy of every message received by P. Then the axiom states

$$\forall P : \gamma_P \subseteq \delta_P$$

The multiset difference operator $\ominus$ is capable of modelling both undelivered messages and out-of-order messages. Lost messages, for instances, remain forever in $\delta_P \ominus \gamma_P$. Out-of-order messages are selected (nondeterministically) for selection from unordered multiset $\delta_P \ominus \gamma_P$. We now define the send and receive axioms as follows

Definition 12 (Send axiom)**.** The statement

$$\textbf{send}\ expr\ \textbf{to}\ P$$

is same as executing the assignment

$$\delta_P = \delta_P \oplus expr$$

where $\delta_P \oplus expr$ is the multiset containing elements of δ_P with the addition of a value $expr$. The axiom states

$$\{X^{\delta_P}_{\delta_P \oplus expr}\}\ \textbf{send}\ expr\ \textbf{to}\ P\ \{X\}$$

where X is the postcondition.

In the send axiom, the execution of the send command begun in the state satisfying $X^{\delta_P}_{\delta_P \oplus expr}$. In this state, the local state of the process is updated with $expr$. As explained before, $X^{\delta_P}_{\delta_P \oplus expr}$ refers to the substitution of all occurrences of δ_P by $\delta_P \oplus expr$. The execution of the send command terminates in a state satisfying X.

Definition 13 (Receive axiom)**.** The statement

$$\textbf{receive}\ m\ \textbf{when}\ \beta$$

makes β true when **receive** terminates. The axiom states

$$\{Y\}\ \textbf{receive}\ m\ \textbf{when}\ \beta\ \{Z \wedge \beta\}$$

where Y is the precondition and $Z \wedge \beta$ is the postcondition. β in addition to Z is a postcondition since the command makes β true when it executes. Z is the part of the postcondition defined on the local state of the process which executes the receive command. For the purpose of proving satisfaction, conditions will be imposed on Z.

Satisfaction rule

Let us consider the **receive** statement

$$r: \ \textbf{receive}\ m\ \textbf{when}\ \beta$$

in process P. In order for statement r to execute and receive a message with value MSG, β^m_{MSG} has to be true and a message with value MSG must have been sent to P but not yet received, i.e. $MSG \in (\delta_P \ominus \gamma_P)$. Thus, the system state just before MSG is assigned to m is

$$pre(r) \ \wedge \ \beta^m_{MSG} \ \wedge \ MSG \in (\delta_P \ominus \gamma_P) \tag{7.1}$$

where $pre(r)$ is the precondition for statement r. Execution of statement r resulting in receipt of message with value MSG is same as executing the multiple assignment statement

$$(m, \gamma_P) = (MSG, \gamma_P \oplus MSG)$$

For this assignment to satisfy the postcondition $Z \wedge \beta$ of the receive axiom, the execution should satisfy the state

$$(Z \wedge \beta)^{m,\gamma_P}_{MSG,\gamma_P \oplus MSG}$$

The auxiliary variable γ_P is not free in β and so we have

$$Z^{m,\gamma_P}_{MSG,\gamma_P \oplus MSG} \ \wedge \ \beta^m_{MSG} \tag{7.2}$$

Therefore, postcondition $post(r)$ will be true when r terminates provided (7.1) $\Rightarrow$ (7.2). That is, when

$$Sat(r): \ (pre(r) \ \wedge \ \beta^m_{MSG} \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \Rightarrow Z^{m,\gamma_P}_{MSG,\gamma_P \oplus MSG} \tag{7.3}$$

is valid. $Sat(r)$ essentially outlines the condition under which the execution of a receive statement will make the assertions in the sequential proof valid.

Process G ::
$\{\delta_P = \emptyset\}$
$\sigma_G = val;$
$\{\delta_P = \emptyset \wedge \sigma_G = val\}$
$s:$ **send** σ_G **to** P;
$\{val \in \delta_P \wedge \sigma_G = val\}$

Process P ::
$\{\gamma_P = \emptyset\}$
$r:$ **receive** m **when** $True;$
$\{m \in \gamma_P\}$
$\vdots$
$\Phi_D : \sigma_P + m = k$

Figure 7.1: A two process illustration with no restricted postcondition

Definition 14 (Satisfaction proof)**.** For every **receive** statement r, prove $Sat(r)$ is valid.

7.1.3 Proof sketches

After summarising the axioms and proof rules, we now show their use for a simple two-process message-passing program by adapting the case studies from [113]. The purpose of these proofs is to show how the techniques underlined [113] can be used to prove correctness of distributed computing processes embedded with distributed opaque predicates. We proceed in a "ground-up" manner — meaning that the proofs get progressively complicated with added postconditions and invariants to guarantee that a message sent from one process is delivered in order to its receiver. We also show how the sequential and satisfaction proofs can be extended to apply to a three process scenario.

Let P denote the process which is obfuscated with a distributed opaque predicate Φ_D and G be a *guard* process. We assume that the *guard* updates pointer location on its local copy of the doubly circular linked list (which is denoted by σ_G, the local state of *guard* process G) and sends it to process P. Process P receives σ_G from G and computes Φ_D by adding its local state value σ_P with the received value. This is represented in Figure 7.1. We now show the satisfaction for statement r in P.

Proof. According to 7.3 we have

$$Sat(r): \ (\gamma_P = \emptyset \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \Rightarrow (m \in \gamma_P)^{m,\gamma_P}_{MSG,\gamma_P \oplus MSG}$$

$$\equiv (\gamma_P = \emptyset \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \Rightarrow MSG \in (\gamma_P \oplus MSG)$$

Process G ::
$\{\delta_P = \emptyset\}$
$\sigma_G = val$;
$\{\delta_P = \emptyset \wedge \sigma_G = val\}$
$s:$ **send** σ_G **to** P;
$\{val \in \delta_P \wedge \sigma_G = val\}$

Process P ::
$\{\gamma_P = \emptyset\}$
$r:$ **receive** m **when** $True$;
$\{m = val\}$
$\vdots$
$\Phi_D : \sigma_P + m = k$

Figure 7.2: A two process illustration with restricted postcondition

This is true since any message received will satisfy the postcondition. □

But, the postcondition $\{m \in \gamma_P\}$ is not enough to guarantee that m has a value val in the predicate $\Phi_D : \sigma_P + m = k$. If we tighten the postcondition of r, the sequential proof of Figure 7.2 is obtained. Is $Sat(r)$ still valid?

Proof. By 7.3 we have

$$Sat(r): \ (\gamma_P = \emptyset \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \Rightarrow (m = val)^{m,\gamma_P}_{MSG,\gamma_P \oplus MSG}$$

$$\equiv (\gamma_P = \emptyset \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \Rightarrow (MSG = val)$$

This is false since some other message may have been received from the buffer. □

Since σ_G is the only message being sent, it is not necessary to strengthen the guard β of process P with $\sigma_G = val$. Instead, Schlichting and Schneider suggested strengthening $pre(r)$. Let the invariant I^1_G of process G be

$$I^1_G : (\forall M : M \in \delta_P : M = val)$$

meaning all messages sent from G will have a value val. Let us see if $Sat(r)$ is valid

Proof.

$$Sat(r): \ (\gamma_P = \emptyset \ \wedge \ I^1_G \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \Rightarrow (m = val \ \wedge \ I^1_G)^{m,\gamma_P}_{MSG,\gamma_P \oplus MSG}$$

Process G ::
$\{\delta_P = \emptyset \wedge \gamma_G = \emptyset\}$
$\sigma_G = val_1$;
$\{\delta_P = \emptyset \wedge \gamma_G = \emptyset \wedge \sigma_G = val_1\}$
s : **send** σ_G **to** P;
$\{val_1 \in \delta_P \wedge \gamma_G = \emptyset \wedge \sigma_G = val_1\}$
r' : **receive** n **when** $n = ACK$;
$\{ACK \in \gamma_G\}$
$\sigma_G = val_2$;

Process P ::
$\{\gamma_P = \emptyset \wedge \delta_G = \emptyset\}$
r : **receive** m **when** $True$;
$\{m = val_1 \wedge val_1 \in \gamma_P \wedge \delta_G = \emptyset\}$
$\vdots$
$\Phi_D : \sigma_P + m = k$
$\vdots$
$\{m = val_1 \wedge val_1 \in \gamma_P \wedge \delta_G = \emptyset\}$
s' : **send** ACK **to** G;
$\{ACK \in \delta_G \wedge val_1 \in \gamma_P\}$

Figure 7.3: A two process illustration with an acknowledgement message

$$\equiv (\gamma_P = \emptyset \ \wedge \ (\forall M : M \in \delta_P : M = val) \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \Rightarrow (MSG = val)$$

This is true since all messages have the value val. □

Clearly, this is not an interesting case since the *guard* process is only allowed to send messages with value val. By allowing an acknowledgement message (ACK) to be sent back from process P, the *guard* process G can be allowed to have more flexibility in changing its local state σ_G as shown in the sequential proof of Figure 7.3.

The role of the acknowledgement message is to ensure that the local state σ_G of G does not change until it is guaranteed that P receives the message. This is the reason why special acknowledgement messages of type `RESPONSE` were introduced in the system (refer to 6.4). Again, following Schlichting and Schneider, we give the satisfaction proofs of two receive statements r and r'. Let us see if $Sat(r')$ is valid

Proof.

$$Sat(r') : \ (val_1 \in \delta_P \ \wedge \ \gamma_G = \emptyset \ \wedge \ \sigma_G = val_1 \ \wedge \ MSG = ACK \ \wedge \ MSG \in (\delta_G \ominus \gamma_G))$$
$$\Rightarrow (ACK \in \gamma_G)^{n,\gamma_G}_{MSG,\gamma_G \oplus MSG}$$
$$\equiv (val_1 \in \delta_P \ \wedge \ \gamma_G = \emptyset \ \wedge \ \sigma_G = val_1 \ \wedge \ MSG = ACK \ \wedge \ MSG \in (\delta_G \ominus \gamma_G))$$
$$\Rightarrow ACK \in (\gamma_G \oplus MSG)$$

This is true since the guard for statement r' $(n = ACK)$ ensures that only a message n with value ACK is received. □

Let us see if $Sat(r)$ valid

Proof.

$$\begin{aligned} Sat(r): \ & (\gamma_P = \emptyset \wedge \delta_G = \emptyset \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \\ \Rightarrow & (m = val_1 \wedge val_1 \in \gamma_P \wedge \delta_G = \emptyset)^{m,\gamma_P}_{MSG,\gamma_P \oplus MSG} \\ \equiv & (\gamma_P = \emptyset \wedge \delta_G = \emptyset \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \\ \Rightarrow & (m = val_1 \wedge val_1 \in (\gamma_P \oplus MSG)) \end{aligned}$$

This is false since the precondition is not sufficient to guarantee that the postocondition of a message $m = val_1$ is received. □

Schlichting and Schneider suggested strengthening the assertions based on an invariant constructed with the following rule:

$$I : Message\ not\ sent \ \vee Acknowledgement\ received \ \vee Local\ state\ True$$

This will ensure that the *guard* process G does not change its local state value σ_G from val_1 to val_2 until after process P has finished its use of val_1. For our simple example, an invariant I^2_G of the *guard* process G can be defined as

$$I^2_G : \delta_P = \emptyset \vee \sigma_G = var_1 \vee ACK \in \gamma_G$$

Let us see if combining I^2_G with I^1_G, we get a valid satisfaction formula

Proof.

$$\begin{aligned} Sat(r): \ & (\gamma_P = \emptyset \wedge \delta_G = \emptyset \ \wedge \ I^1_G \ \wedge \ I^2_G \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \\ & \Rightarrow (m = val_1 \wedge val_1 \in (\gamma_P \oplus MSG) \wedge I^1_G \wedge I^2_G) \end{aligned}$$

Process $G1$ ::
$\sigma_{G1} = val_1$;
$\{\gamma_{G1} = \emptyset \wedge \sigma_{G1} = val_1\}$
$s1$: **send** σ_{G1} **to** P;
$\{val_1 \in \delta_P \wedge \sigma_{G1} = val_1\}$

Process $G2$ ::
$\sigma_{G2} = val_2$;
$\{\gamma_{G2} = \emptyset \wedge \sigma_{G2} = val_2\}$
$s2$: **send** σ_{G2} **to** P;
$\{val_2 \in \delta_P \wedge \sigma_{G2} = val_2\}$

Figure 7.4: Two guard processes $G1$ and $G2$ sending messages to P in Figure 7.5

This is true and will guarantee the satisfaction of the **receive** statement r. □

These simple examples with sequential and satisfaction proofs demonstrate the necessary conditions (assertions and invariants) under which message-passing programs with distributed opaque predicates can be proved correct. With three or more processes and multiple **send** and **receive** commands, the satisfaction proofs with acknowledgement messages would be expanded considerably with the addition of more pre and postconditions but the basic proof structure should be similar.

Next we consider a three-process message exchange scenario now. In the Figure 7.4, two guard processes $G1$ and $G2$ send one message each to the process P. Process P, in Figure 7.5, receives two messages from $G1$ and $G2$ and tries to evaluate the distributed opaque predicate Φ_D at locations $l1$ and $l2$ based on two messages received at $r1$ and $r2$ respectively. Let us assume that Φ_D correctly evaluates (predetermined obfuscator value) at $l1$ only if the message received at $r1$ is the one sent from process $G1$. Similarly, Φ_D correctly evaluates at $l2$ only if the message received at $r2$ is the one sent from process $G2$. For simplicity, we do not consider acknowledgement messages. Since the system under consideration is asynchronous, the receive order of messages cannot be guaranteed. We will see if the satisfaction proofs $Sat(r1)$ and $Sat(r2)$ hold.

First we notice that the **send** preconditions for both $G1$ and $G2$ do not contain $\{\delta_P = \emptyset\}$ since neither of the two processes could have sent a message to P before statements $s1$ and $s2$ are executed. Let us check the validity of $Sat(r1)$ in process P.

Process P ::
$\{\gamma_P = \emptyset\}$
$r1:$ **receive** m **when** $True$;
$\{m = val_1 \wedge val_1 \in \gamma_P\}$
$\vdots$
$l1 : \Phi_D : \sigma_P + m = k$
$\vdots$
$\vdots$
$r2:$ **receive** m **when** $True$;
$\{m = val_2 \wedge val_2 \in \gamma_P\}$
$\vdots$
$l2 : \Phi_D : \sigma_P + m = k$
$\vdots$

Figure 7.5: Process P receiving messages from $G1$ and $G2$ in Figure 7.4

Proof. According to 7.3 we have

$$Sat(r1): \ (\gamma_P = \emptyset \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \Rightarrow (m = val_1 \wedge val_1 \in \gamma_P)^{m,\gamma_P}_{MSG,\gamma_P \oplus MSG}$$

$$\equiv (\gamma_P = \emptyset \ \wedge \ MSG \in (\delta_P \ominus \gamma_P)) \Rightarrow (m = val_1 \wedge val_1 \in (\gamma_P \oplus MSG))$$

This is false since the precondition is not strong enough to guarantee that the message received has a value val_1 and not val_2. □

The proof for $Sat(r2)$ is similar. Trivially, we can prove both $Sat(r1)$ and $Sat(r2)$ true by making $val_1 = val_2$. Let us see if we can strengthen the precondition of $r1$ with invariants. For two-process illustrations, we recollect that invariant I^1_G of process G was

$$I^1_G : (\forall M : M \in \delta_P : M = val)$$

Since none of the statements of process P interfered with I^1_G, it was an invariant of the entire concurrent program $[G \parallel P]$. However,

$$I^3_{G1} : (\forall M : M \in \delta_P : M = val_1)$$

Process P ::
$\{\gamma_P = \emptyset\}$
$r1:$ **receive** m **when** $m = val_1$;
$\{m = val_1 \wedge val_1 \in \gamma_P\}$
$\vdots$
$l1 : \Phi_D : \sigma_P + m = k$
$\vdots$
$\vdots$
$r2:$ **receive** m **when** $m = val_2$;
$\{m = val_2 \wedge val_2 \in \gamma_P\}$
$\vdots$
$l2 : \Phi_D : \sigma_P + m = k$
$\vdots$

Figure 7.6: Process P with strengthened guards

cannot be an invariant for $G1$ in case of our three-process illustration. This is because it interferes with the **send** statement $s2$ of process $G2$. The only other strategy is to strengthen the guards of **receive** statements $r1$ and $r2$ as shown in Figure 7.6. The guards $m = val_1$ and $m = val_2$ guarantee that only messages with values val_1 and val_2 are received at statements $r1$ and $r2$ respectively. In Section 6.4, the role of the vector clocks is to help the obfuscated process select a message with "expected" value from the buffer. Since vector clocks enforce causality, it is relatively straightforward to determine the precedence of messages.

7.2 Attacking distributed opaque predicates

We stated in Section 6.3 that static and dynamic analysis attacks on distributed opaque predicates would not reveal their outcome because the invariant communication patterns responsible for maintaining them are nondeterministic in nature. In this section, we evaluate the obfuscatory strength of distributed opaque predicates by following the attack process described in Chapter 3 Section 3.2. But, before describing the attack process on distributed opaque predicates, we define a successful attack on an opaque predicate as

follows:

Definition 15 (Opaque Predicate Attack)**.** An attack on a process P containing an opaque predicate Φ is considered to be *successful* if an adversary is able do any of the following:

1. If the predicate is Opaquely True (Φ^T) then the conditional statement involving the predicate could be substituted with the corresponding "true branch" such that the I/O behaviour is preserved for every execution of P.

2. If the predicate is Opaquely False (Φ^F) then the conditional statement involving the predicate could substituted with the corresponding "false branch" such that the I/O behaviour is preserved for every execution of P.

3. If the predicate is Opaquely Unknown ($\Phi^?$) then the conditional statement involving the predicate could be substituted with either its "true branch" or "false branch" such that the I/O behaviour is preserved for every execution of P.

We recollect from Section 3.2 that an attack process essentially consists of four steps. In the first step of this process, candidate programs are chosen to be obfuscated with the transform which is being tested for its obfuscatory strength. We consider two target applications P_1 and P_2 for mounting the attacks. P_1 is a simple single-method procedural C program while P_2 is a concurrent object-oriented Java application which runs in a distributed mobile agents environment. We considered two candidate programs for the purpose of comparing the difficulty of attacking a distributed opaque predicate against a simple opaque predicate. As the second step of the attack process, P_1 is obfuscated with an opaque predicate whereas P_2 is obfuscated with a distributed opaque predicate. It is worth nothing that both these obfuscations are applied at the source code level of the programs. In the following subsections, we describe each of our target applications briefly.

```
int OpaquelyTrue()
{
  int n = 10;
  int i = 0;
  int x = 0;
  int y = 1;
  while (i < n)
  {
    i++;
    if (i < 5 || x < y)
    x = x + i;
    else y = x * i;
    y = y * i;
  }
  out(x);
  out(y);
}
```

Figure 7.7: P_1: Sum/Product example with an Opaquely True predicate

Target application P_1: Sum/Product example

The Sum/Product example from Chapter 4 is show Figure 7.7. In the method `OpaquelyTrue`,

$$\Phi^T : (i < 5 \;||\; x < y)$$

is an Opaquely True predicate since it always evaluates to True [40]. We have used an invariant to construct this predicate. Since, $x = sum(1..i)$ and $y = prod(1..i)$ then for $i \geq 4$ we have $x < y$. Immediately before this expression we have $i++$ and so we need to have $i < 5$ rather than $i < 4$. Therefore, the "false branch" **else** $y = x * i$; of the method `OpaquelyTrue` is never taken.

Target application P_2: e-Commerce example

Our second target application P_2 is a three-process distributed computing environment simulated with mobile agents. We embedded a distributed opaque predicate Φ_D in one of the agents and simulated a set of message passing acts between them so that the values of Φ_D are influenced by **send** and **receive** of messages. Before we describe P_2 in details,

we briefly elucidate the mobile agents middleware in which we spawn the agents. Until otherwise distinguished, we use the terms process and agents interchangeably.

We selected the JADE (Java Agents Development Environment) mobile agents platform for conducting our experiments [71]. JADE is a middleware that facilitates the development of multi-agent systems. It includes:

- A runtime environment where JADE agents can "live" and that must be active on a given host before one or more agents can be executed on that host.
- A library of classes that programmers have to/can use (directly or by specialising them) to develop their agents.
- A suite of graphical tools that allows administrating and monitoring the activity of running agents.

In JADE, each host has an agent container, to hold its local agents, while each agent is an active thread, with its own behaviour which is a module in an agent code that describes all the actions to be carried out by the agent in its lifetime. An agent can execute several behaviours concurrently. JADE provides a virtual agent platform, by which all agents can interact with each other, without considering their hosts or containers. Agents in JADE are executed concurrently, while all of their interactions are under control of the JADE platform. However, the scheduling of behaviours in an agent is not preemptive (as for Java threads) but cooperative. This means that when a behaviour is scheduled for execution its *action()* method is called and runs until it returns. Therefore it is the programmer who defines when an agent switches from the execution of a behavior to the execution of the next one. The agent thread path of execution in JADE is shown in the Figure 7.8.

P_2 is an electronic commerce scenario built using mobile agents in the JADE platform [86]. The scenario in our particular case is a stock market transaction system (such as one in [15]). In this application, mobile agents act as `PriceSetters` or `StockTraders`. A query in form of a predicate denoting certain stock prices that need to be monitored

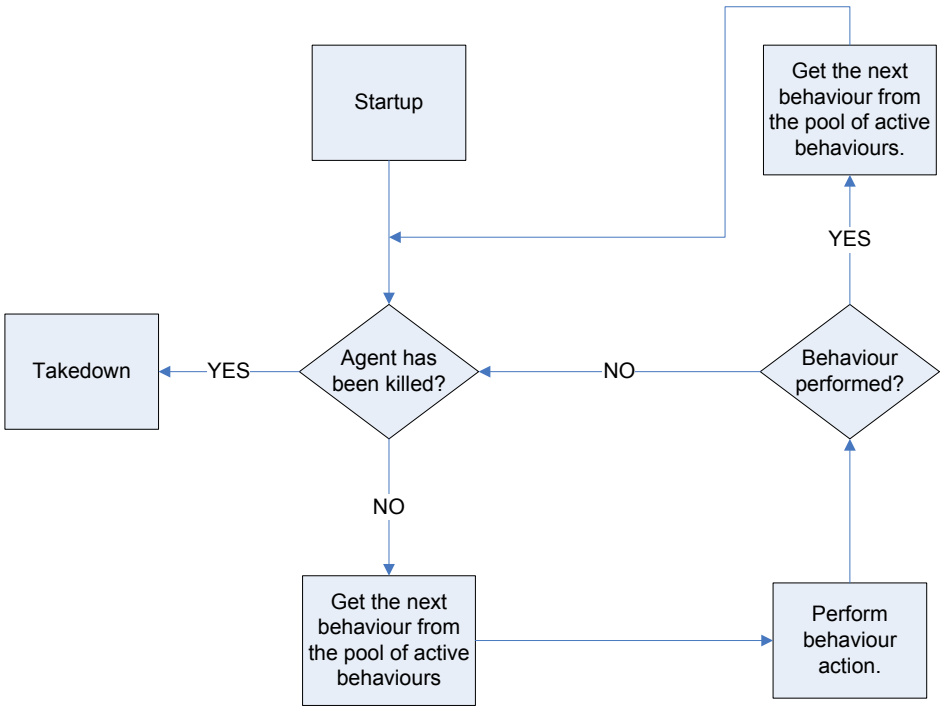

Figure 7.8: Agent thread path of execution

can be entered. The monitor agent *MA* evaluates the query gives notification when the query is satisfied. Thus, *MA* is the process of interest to the adversary $\mathcal{A}$. Lets say a `StockTrader` agent wants to trade stock according to some strategy (for e.g., if Cisco is below $10 and IBM more than $100 then the trader sells IBM shares and buys Cisco shares. In the e-commerce scenario, the `PriceSetter` agents set the prices of different stocks. Figure 7.9 gives the UML class diagram of the stock trading system.

In our simple implementation, we have effectively run detection algorithm with predicates that are formed over a conjunction of local states. In this scheme, the global predicate is broken into a conjunction of local predicates and passed on to individual agents for satisfaction. The function `checkPredicate()` is invoked so as to report to the monitor agent that the corresponding local predicates have become true. Since the agent ID is only known at compile time, the monitor must rely on JADE directory service for search agents with a certain properties. Upon registration, the monitor will start receiving reports from the monitored agents when their local predicates are satisfied. These reports are used in the detection algorithm to detect the occurrence of the global predicate. The

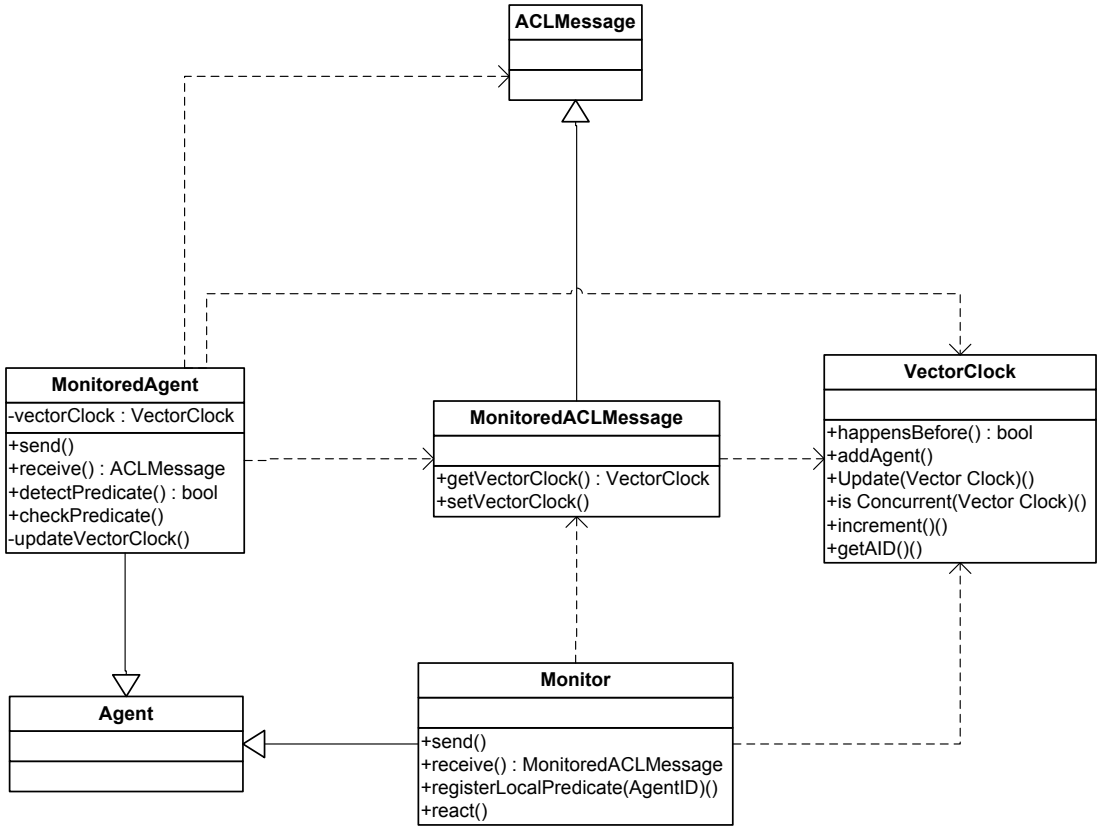

Figure 7.9: UML class diagram of the stock trading system depicting target application P_2

design also implements a vector clock to maintain causality of messages and events in the system. The vector clock is marshalled with the ACL (Agent Communication Language) messages and received/sent by the agents against each communication act.

Before describing our attack on two target applications P_1 and P_2, we make the following assumption:

Assumption 1 The attacker $\mathcal{A}$ has powerful pattern matching tools in order to identify predicates in applications P_1 and P_2 respectively. That is,

$$id(P_1) = \Phi \text{ and } id(P_2) = \Phi_D \text{ are successful}$$

This assumption is quite strong since pattern matching an opaque predicate is a non-trivial task especially if the predicate is designed to be stealthy. However, the main objective of this evaluation is not on pattern matching tools/techniques but to show

that a current state-of-the-art slicing tool is incapable of finding a distributed opaque predicate. We therefore, make this assumption to help us proceed to the next step of the four-step attack process described in Section 7.2. If an adversary is unable to identify opaque predicates from normal program conditionals, then the attack becomes all the more difficult since the four-step attack process of Section 7.2 has to be repeated for every conditional statement that is suspected to be an opaque predicate.

The next section is dedicated for describing the attack model since it is the most important component of the attack process. We describe the steps of the attack model and then describe each of these steps in details. For each of these steps, we also describe the interpretation of the output as seen by the human attacker — this is the last step of the attack process.

7.3 The attack model

The third step of the attack process consists of applying an "attack model" to the obfuscated programs P_1 and P_2. The attack model performs attack in four steps. The first step "identifies" a predicate.

Definition 16 (Identify)**.** Identification of an opaque predicate Φ in a process P involves distinguishing Φ from other predicates in P. Thus,

$$id(P) = \Phi$$

The second step of the attack "slices" P based on Φ.

Definition 17 (Slice)**.** Slicing of a process P with respect to the opaque predicate Φ involves finding an executable slice $\mathcal{S}$, where $\mathcal{S}$ takes the slicing criterion as its input and produces 1 bit of output. Here, the slicing criterion includes the program to be sliced, the program point where opaque predicate is defined, and the variables involved in the predicate. Thus,

$$slice(P, \Phi, \mathcal{V}(\Phi)) = \mathcal{S}_\Phi$$

It is worth nothing that the definition of a *slice*() is similar to our slicing notation in Section 4.1.3. The slice is computed on P from the occurrence of the predicate Φ with respect to the set of variables $\mathcal{V}(\Phi)$ in Φ. The next step towards a successful attack is to "evaluate" Φ.

Definition 18 (Evaluate)**.** Evaluation of an opaque predicate Φ in process P involves finding out whether Φ holds at the point of its occurrence for every execution of P. Thus,

$$eval(\mathcal{S}_\Phi) = \{0, 1\}$$

We have three cases

$$eval(\mathcal{S}_\Phi) = \left\{ \begin{array}{l} 1 \text{ if } \Phi \in \Phi^T \\ 0 \text{ if } \Phi \in \Phi^F \\ 0 \text{ or } 1 \text{ if } \Phi \in \Phi^? \end{array} \right\} \text{and } eval(\mathcal{S}_\Phi) \text{ terminates for all runs of } \mathcal{S}_\Phi$$

The last step in the attack model involves "substituting" Φ in P.

Definition 19 (Substitute)**.** Substitution of an opaque predicate Φ in process P involves replacing the conditional statement involving Φ with the "true branch" if $eval(\mathcal{S}_\Phi) = 1$ and with the "false branch" if $eval(\mathcal{S}_\Phi) = 0$. Thus,

$$sub(eval(\mathcal{S}_\Phi), \Phi, P) \rightarrow P'$$

Having described the attack model for mounting a successful attack on an opaque predicate Φ in a process P, we now proceed to define the human attacker in the context. The attack will apply the attack model on the obfuscated programs and analyse their outputs.

Definition 20 (Attacker)**.** An attacker $\mathcal{A}$ with respect to a target process P and an opaque predicate Φ is characterised by the following three attributes:

Goal The goal of $\mathcal{A}$ is to convert the target program P into its deobfuscated equivalent P' such that the opaque predicate Φ is substituted with its corresponding true or false branch and P' has the same I/O behaviour as P for all possible executions. Thus,

$$\mathcal{A}(P, \Phi) = P' \text{ where } P' = P \backslash \Phi$$

Strategy $\mathcal{A}$ can perform four attack steps in the attack model, viz. identify, slice, evaluate, and substitute, in sequence.

Limitations $\mathcal{A}$ is allowed to perform only these four attack steps in sequence. To identify, $\mathcal{A}$ can use any pattern matching tool and observational heuristics. To slice, $\mathcal{A}$ is restricted to use either of the two static slicers Indus or CodeSurfer (discussed in Chapter 3 and Chapter 4 respectively). For the purpose of evaluating Φ, $\mathcal{A}$ is allowed to repeatedly execute $\mathcal{S}_\Phi$. To substitute, $\mathcal{A}$ is allowed to use any available tool for replacing Φ with its corresponding true or false branch.

The limitations on adversarial powers of $\mathcal{A}$ are not unreasonable and do not overly restrict our adversary. We have given $\mathcal{A}$ the state-of-the-art analysis tools to mount the attacks on P and it follows from logical reasoning that all fours steps are necessary for mounting a successful attack — for instance, a slice with respect to the opaque predicate $\mathcal{S}_\Phi$ is needed to find the range of values $\mathcal{V}(\Phi)$ can take during the evaluation step. It is worth noting that simple observation of the values an opaque predicate takes during different program runs is not enough to evaluate whether the predicate is opaquely unknown. This is because, an opaquely unknown predicate may evaluate to true (false) for the first n runs and then evaluate to a false (true) at the $n+1^{th}$ execution of the program due to underlying nondeterminism of the execution environment.

7.3.1 Slicing attack

Under the assumption that the attacker $\mathcal{A}$ is able to identify opaque predicate Φ in P_1 and distributed opaque predicate Φ_D in P_2, we now proceed to mount slicing attack on

```
int OpaquelyTrue()
{
  int n = 10;
  int i = 0;
  int x = 0;
  int y = 1;
  while (i < n)
  {
    i++;
    if (i < 5 || x < y)
    x = x + i;
    else y = x * i;
    y = y * i;
  }
}
```

Figure 7.10: P_1 under slicing attack using Φ as slicing criterion

the target applications.

Slicing attack on P_1

We use CodeSurfer to mount slicing attack on P_1. Using Φ as the slicing criterion, we obtain the slice $\mathcal{S}_\Phi$ of Figure 7.10. We can see that $\mathcal{S}_\Phi$ is a correct executable subset of P_1 which $\mathcal{A}$ can use to evaluate the value of Φ.

Slicing attack on P_2

In order to find $\mathcal{S}_{\Phi_D}$ with distributed opaque predicate Φ_D of target application P_2 as the slicing criterion, the attacker $\mathcal{A}$ needs to slice parts of the agents' code which could affect the value of Φ_D at obfuscated control-flow points of MA. Slicing of distributed programs is a major challenge due to the timing related interdependencies among processes. Moreover, the slicer must rely on alias analysis on dynamic data structures [54, 59] to determine the exact values of local states the cooperating agents can take. Additionally, the slicer must rely on inter-process escape analysis to determine the objects that can be referenced in processes separate from the ones in which they are allocated [135]. Although much research work on alias analysis and escape analysis have been done in the last few

years for procedural and object-oriented programs, we have been unsuccessful in finding a slicer which can perform alias analysis on programs that communicate by asynchronous message-passing. The reason why this problem has not been addressed by the program analysis community is because we do not have efficient, precise and scalable algorithms for performing simpler cases of alias analysis in multi-threaded programs yet. Asynchronous concurrent programs present problems that are much more intractable.

Even so, we present a slicing attack using the Kaveri Eclipse plugin of the state-of-the-art concurrent Java slicer Indus. A screenshot from Eclipse IDE, given in Figure 7.11, shows that Indus throws an exception while slicing the mobile agent code of application P_2. Thus, an attacker using Indus will fail to slice the agents' code and consequently will not be able to find local states that influence the outcome of Φ_D in MA. Indus fails to slice because it is incapable of tracking contexts that escape via asynchronous messages to other processes. To the best of our knowledge, an implementation of dynamic message-passing slicer does not exist yet. Li *et al.* in [75] describe an algorithm for slicing message-passing programs. Considering that the idea can be engineered to implement a working dynamic message-passing slicer in the future, we argue in the next subsection that building the global state by combining all possible local states of individual agents will still be hard for the attacker.

7.3.2 Evaluation attack

Given the assumption that the attacker $\mathcal{A}$ can repeatedly execute the slice $\mathcal{S}_\Phi$, we now proceed to show how $\mathcal{A}$ can mount evaluation attack.

Evaluation attack on P_1

After obtaining a slice $\mathcal{S}_\Phi$ of the application P_1 through slicing attack, the attacker $\mathcal{A}$ can observe the values taken by the opaque predicate Φ by tracing the slice $\mathcal{S}_\Phi$ for all iterations of the variable i. The Table 7.1 shows the first ten iterations of variable i. We can see from the Table, for the first four iterations of variable i, $(i < 5)$ is true and

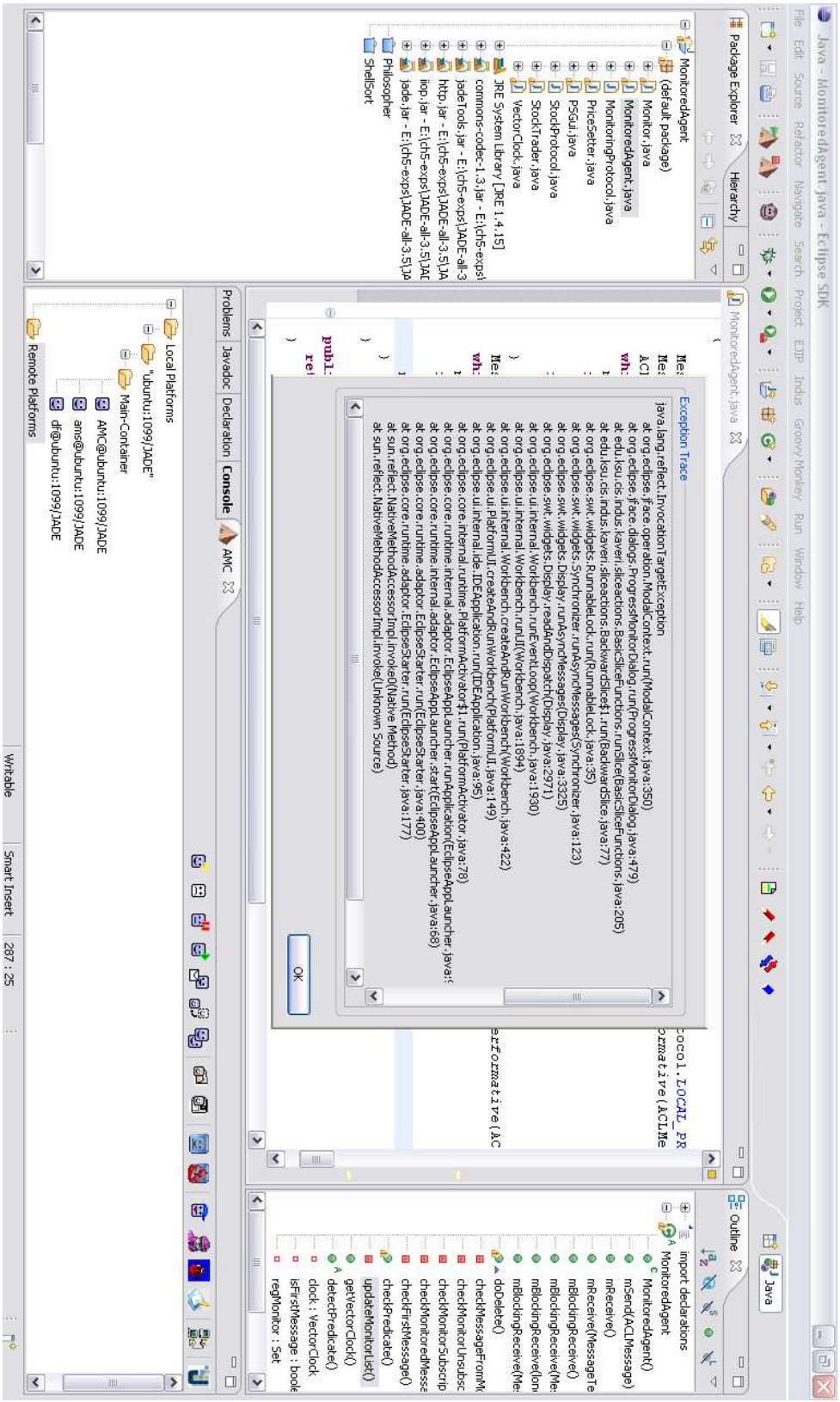

Figure 7.11: P_2 under slicing attack using Φ_D as slicing criterion

i	x	y	$(i < 5)$	$(x < y)$	Φ
1	1	1	T	T	T
2	3	2	T	F	T
3	6	6	T	F	T
4	10	24	T	F	T
5	15	120	F	T	T
6	21	720	F	T	T
7	28	5040	F	T	T
8	36	40320	F	T	T
9	45	362880	F	T	T
10	55	3628800	F	T	T

Table 7.1: Evaluation trace for Φ for P_1

$(x < y)$ is false. However, as the predicate is expressed as the disjunction of $(i < 5)$ and $(x < y)$, it is true. However, for all positive iterations greater than four, $(i < 5)$ is false and $(x < y)$ is true. However, Φ is still true since it is a disjunction of the terms $(i < 5)$ and $(x < y)$. Thus, predicate Φ is found out to be Opaquely True for all positive values that variable i can take.

Evaluation attack on P_2

For an evaluation attack on P_2 to succeed, the following two assumptions need to be satisfied:

Assumption 2 The attacker $\mathcal{A}$ must have successfully carried out slicing attack on P_2.

Assumption 3 The attacker $\mathcal{A}$ can monitor the communication events using an external *sniffer* agent in order to observe the individual local states formed by MA and other agents in P_2. That is, $\mathcal{A}$ using the sniffer agent has access to the send multiset δ_i for every agent i in P_2.

Assumption 2 needs to be satisfied before an evaluation attack can be mounted because portions of the local states of processes involved in maintaining the distributed opaque predicate has to be found out. As we argued earlier, simple monitoring of predicate values at different program runs will not help in identifying opaquely unknown predicates since a predicate which holds true (false) for the first n runs of the program may become false

(true) in the $n+1^{th}$ run because of underlying program nondeterminism (such as swapping of message order). We argued in the Section 7.3.1 that slicing P_2 is intractable even with the current generation slicer Indus. Assumption 3 makes the relaxation that $\mathcal{A}$ can monitor the communication events using a *sniffer* agent and scenario is depicted in Figure 7.12.

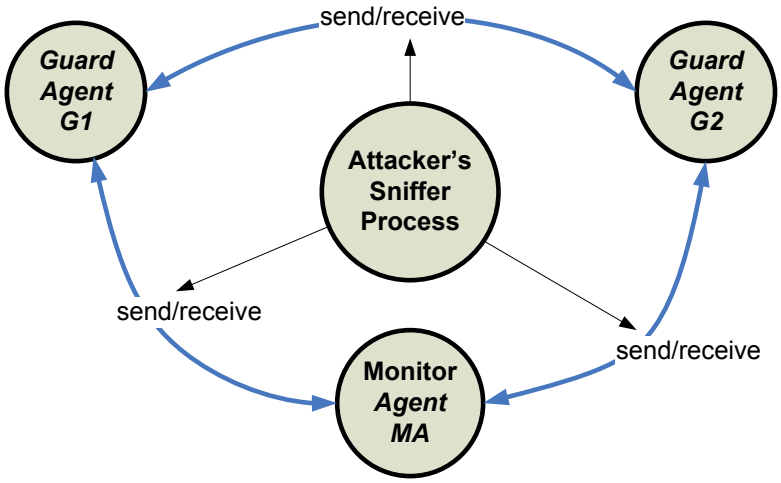

Figure 7.12: An illustration of Assumption 3 where an attacker $\mathcal{A}$ actively monitors state changes to detect the distributed opaque predicate Φ_D in P_2.

But even hypothetically assuming that a slicing attack can be successfully carried out with next generation slicers, we will argue in remainder of the section that evaluation attack on P_2 would still be hard. The problem of distributed opaque predicate evaluation, from the attacker's perspective, can be stated as evaluating Φ_D as a function of the global state formed by agents in P_2. The domain of such predicates is a boolean valued function formed on the set of all possible cuts from all possible executions of the distributed system.

Therefore, the predicate detection problem can also be defined as identifying a cut in which the predicate evaluates to true. The difficulty associated with detection is the fact that the number of states from any execution may be exponential in the number of agents [29].

In the subsections to follow, we model an evaluation attack by discussing how the attacker $\mathcal{A}$ can use three types of available monitoring techniques to determine the outcome of Φ_D from observed local state values.

Snapshot monitoring

The first option $\mathcal{A}$ has is to solve the global predicate evaluation problem by taking *snapshots*. In this strategy, $\mathcal{A}$ uses the sniffer agent to monitor the communication between MA and other agents in P_2 at some predetermined periodic intervals and then combines all the local states obtained to build the global state. This strategy is called *snapshot* approach after [18]. Since communication within distributed systems incurs latency, the consistent global states thus constructed can only reflect some past state of the system. By the time the snapshots are obtained, conclusions drawn about the system by evaluating Φ_D may have no bearing to the present. Therefore, the snapshot algorithm is suitable for monitoring predicates that do not change value throughout the entire program run and since our distributed opaque predicates are temporally unstable in nature, the attacker will not be able to deduce a correct reasoning about the our predicate's behaviour using this algorithm — Φ_D may have held even if it is not detected. It is worth noting that in dedicated high speed network links, such as supercomputing interconnection networks, where latency is of least concern, a snapshot algorithm can be effectively used by an adversary for detecting the outcome of distributed opaque predicates. However, in our model of distributed computation, we assume the mobile agents to be running across remotely located loosely coupled nodes (as in e-commerce environments) and therefore the issue of latency always need to be taken into account while monitoring predicate values.

Passive monitoring

Through the second approach, $\mathcal{A}$ can collect all local state values from individual agents and check for consistent cut using *passive* monitoring [29]. In order to implement this algorithm, the attacker's sniffer agent must monitor the agents in P_2 for portions of their local states that are referenced in $\mathcal{S}_{\Phi_D}$ (assuming this is available by Assumption 2). The sniffer maintains sequences of these local states, one sequence per agent, and uses them to construct the global state. This procedure is based on incrementally constructing the

lattice of consistent global states associated with the distributed computation. The state lattice formed is linear in the number of global states, and the number of global states formed is $O(e^n)$ where e is the maximum number of events monitored and n is the number of agents in P_2. For every global state in the lattice, there exists at least one run that passes through it. Hence, if any global state in the lattice satisfies Φ_D, the distributed opaque predicate is satisfied.

The problem with this type of monitoring is that the attacker will end up facing the state explosion problem in the following two scenarios:

1. Spurious local states will be added if agents other than those contributing to the predicate state are considered in the slice $\mathcal{S}_{\Phi_D}$. This can again happen because of two reasons — first, either the slicer used for slicing distributed opaque predicate Φ_D (if such a slicer is implemented in the future), will most likely include local states from other agents if conservative analyses are used. The second reason which can contribute to addition of irrelevant agent states is when an attacker, unable to find a slicer to mount slicing attack, decides to exhaustively consider all agents in the system — even those that are not acting as guards to the process obfuscated with the distributed opaque predicate.

2. Spurious local states will also be added if the wrong kind of messages are considered to update local state values. Note that in our description of state update in Chapter 6 Section 6.4.3, only one kind of message updates local state values of process while two other only update vector clocks but not the local state values. An attacker, while sniffing messages, will erroneously suspect all messages to be updating local state values of processes if he/she is unable to distinguish between message tags.

Hence, if the number of agents in the system is large and a considerable amount of message exchange takes place, the adversary will face the problem of state explosion while trying find a consistent cut by "walking-through" the lattice thus formed [102]. Additionally, if the attacker fails to monitor some of the agent interactions, the amount of concurrency in the form of local state changes will increase and this will, in turn, increase the states of the

lattice. Increase in the number of agents in $\mathcal{S}_{\Phi_D}$ will increase the dimension of the lattice proportionately. If there are more than one distributed opaque predicate in the system, the attacker has to repeat this passive monitoring process to detect the outcome of each of the predicates. The complexity will further increase if agents are spawned dynamically during *MA*'s execution.

We simulate a simple three process message-passing timing diagram in Figure 7.13. For simplicity, we have not distinguished between the `SYSTEM`, `REQUEST` and `RESPONSE` types of messages. Also, the vector clock has been replaced with labels of the form e_i^j, which denotes the jth event of the ith process. Event e_1^2 is a spurious event which possibly causes a state change without send or receive of message. The corresponding lattice has been shown in Figure 7.14. We note that the lattice is 3-dimensional with each dimension corresponding to a process. Along each dimension, an event is marked e_i^j.

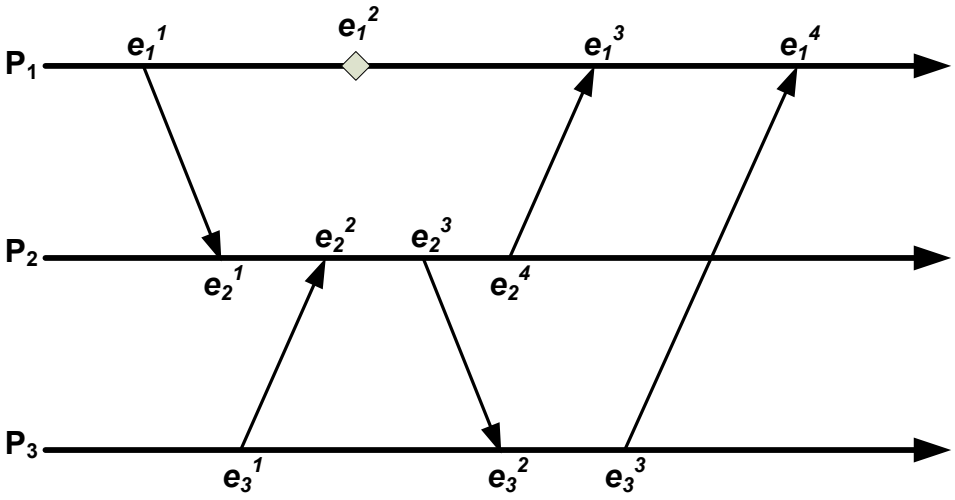

Figure 7.13: A simple three process event diagram. e_i^j indicates the jth event of process i. Event e_1^2 is a spurious event.

Active monitoring

In the final approach, the attacker may choose to use Garg and Waldecker's method [47] for evaluating distributed opaque predicates. Their method exploits the structure of distributed opaque predicate by decomposing it into a conjunction of local predicates and independently evaluating the outcomes of these local predicates. The approach also

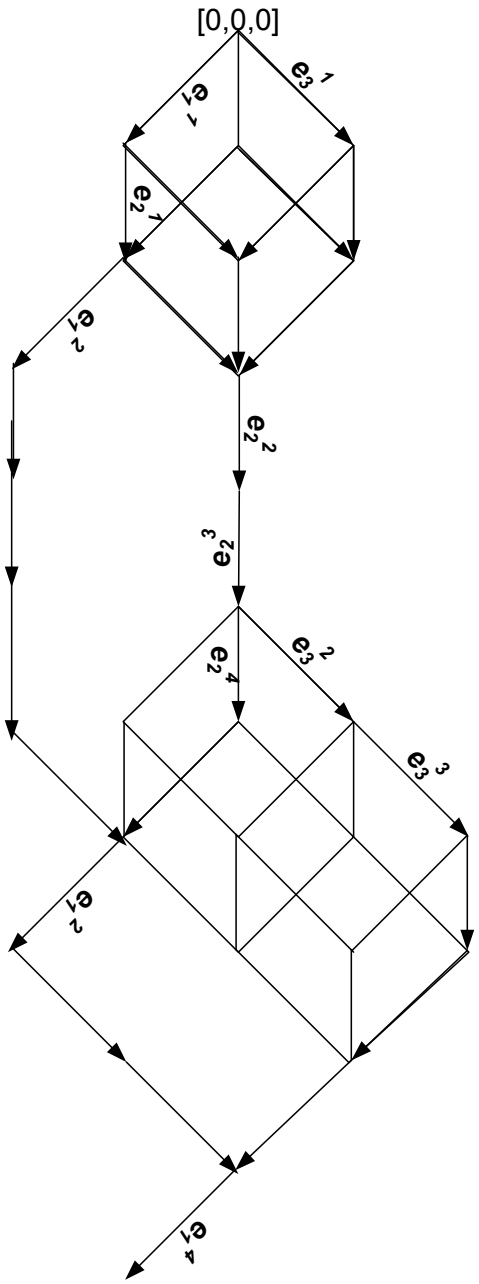

Figure 7.14: The corresponding lattice for Figure 7.13

requires the use of explicit token passing messages between the sniffer agent and the agents in $\mathcal{S}_{\Phi_D}$. This approach works well for predicates that are conjunctive in nature [86]. However, for relational distributed opaque predicates, their method yields no feasible solution because relational predicates cannot be broken down into conjunction of predicates formed on local states. Moreover, the technique requires agents participating in contributing to the value of Φ_D to cooperate with attacker's sniffer agent by main-

taining snapshots (evaluating their component of the predicate) and passing the result and dependence information back to the sniffer agent. This requirement quite unrealistic under reasonable practical assumptions.

Chase and Garg in [20] stated that the evaluation problem is NP-complete even for simple distributed computation where the local states are restricted to take only True or False values and no messages are exchanged within the system. Mittal and Garg in [90] specialised the result for detection of relational predicates in distributed computations. We restate their result in the following theorem:

Theorem 1. Detecting a relational predicate of $\Phi_D = k$ form is NP-complete in general when each of its term can be incremented or decremented by an arbitrary amount at each step.

Proof. The problem is in NP because the general problem of observing an arbitrary boolean expression is in NP. To prove its NP-hardness, an arbitrary instance of the Subset sum problem (refer to [45]) can be reduced to an instance of $\Phi_D = k$. There is a process P_i for each element in the set A of integers that corresponds to the value of the local state σ_i. The initial value of σ_i is set to zero. Each process has exactly one event e_i. The final value of each σ_i after executing e_i is $s(a_i)$, where size $s(a_i) \in Z^+$ for each $s(a_i) \in A$. Finally, k can be set to the sum B. The reduction takes polynomial time and the required subset exists if $\Phi_D = k$ holds. □

We conclude this subsection by observing that out of these three available evaluation approaches, the attacker has to resort to using only the second approach because the other two available approaches are suitable for evaluating predicates that are either constant during the entire program run or can be broken down into a conjunction of local predicates. However, the second approach will be intractable if a large number of agents are used or are spawned dynamically during execution of P_2. Also, in the absence of precise static analysis methodologies, spurious events and state changes would be erroneously taken into consideration by the attacker and this would result in state explosion problem. Under pragmatic assumptions, we believe that practical distributed systems will employ a large

```
int AttackedMethod()
{
  int n = 10;
  int i = 0;
  int x = 0;
  int y = 1;
  while (i < n)
  {
    i++;
    x = x + i;
    y = y * i;
  }
  out(x);
  out(y);
}
```

Figure 7.15: P_1 after Φ is substituted

number of processes as *guards* and hence processes obfuscated with distributed opaque predicates will be resilient to passive monitoring evaluation attacks.

7.3.3 Substitution attack

The last step towards mounting a successful distributed opaque predicate attack is to substitute an Opaquely True predicate with its corresponding "true branch" and an Opaquely False predicate with its corresponding "false branch. This is relatively easy for P_1 with opaque predicate Φ and we show this in Figure 7.15. The method `AttackedMethod` represents the successfully attacked code with the opaque predicate Φ substituted. Since Φ is opaquely true, the substituted code contains only the "true branch" of Φ. However, a substitution attack cannot be mounted for the distributed opaque predicate Φ_D since we demonstrated how the attacker fails to mount both slicing and evaluation attack on P_2.

7.4 Conclusion

The first contribution of this chapter was in illustrating the use of a relaxed CSP framework for proving correctness of asynchronous message-passing programs. We then illus-

trated existing proof rules for proving satisfaction of postcondition of **receive** statements and adapted them for simple two and three process distributed programs. We argued that for a distributed opaque predicate Φ_D, constructed with local states of *guard* processes and the obfuscated process P, to evaluate correctly in P, the *guard* processes must not change their local states unless an acknowledgement from P has been received. Also, for three or more processes, a message ordering scheme by means of vector clocks is needed for correct execution of the system.

Our second contribution is in giving a definition of an opaque predicate attack and observing that an adversary needs to undergo four steps in order to mount a successful attack. We illustrated the four step attack on two applications — the first one a simple single-method procedural program and the second one an electronic commerce scenario simulated with mobile agents in a distributed computing environment. We used state-of-the-art C slicer CodeSurfer and the Java slicer Indus, to slice the corresponding C application and the mobile agents' code respectively. We observed that the first application could be sliced relatively easily whereas Indus was incapable of slicing concurrent programs which communicated by asynchronous message-passing. We then hypothetically argued that even if a message-passing slicer was available in the future, the attacker would find the problem of finding a cut through the global state of the system intractable (exponential in the number of local state changes and message exchanges within the distributed system). Thus, the attack on single-method procedural application was a success but the attack on mobile agents distributed computing application obfuscated with distributed opaque predicate was a failure. Although our proof-of-concept experiments for evaluating the obfuscatory strength of distributed opaque predicates were limited by the lack of appropriate program analysis tools, we conclude that our observations (similar to Wang's observation in her PhD thesis [132]) give a promising indication on the limitations of the powers of an attacker equipped with state-of-the-art attack tools.

It is worth noting that the relational structure of distributed opaque predicates will make these constructs stealthy to an attacker since predicates of this form frequently appear as conditionals in most programs. Moreover, the *guards*, along with the pro-

cess/agent to be obfuscated, maintain the distributed opaque predicate invariant through an embedded communication pattern. This pattern is generated by **send**/**receive** calls embedded within the process' code. Processes in loosely-coupled distributed systems, as such, communicate using message-passing and hence the presence of additional *guards* and their interactions with the host will be unsuspecting from the perspective of an attacker. As far as the performance issues are concerned, distributed systems are inherently loosely-coupled in nature and do not enforce hard timing requirements. Hence, the overall slowdown in system effectuated by additional *guards* and message exchanges might be acceptable to applications having stringent security requirements.

8

Concluding Remarks and Future Work

The field of software obfuscation witnessed a dearth of research results since Barak published his landmark paper in 2001. We believe the main reason for this is the lack of proper understanding of the security requirements for obfuscation. Although it has been proven theoretically that a complete blackbox security for obfuscation cannot be guaranteed, software developers still bank on obfuscation for protecting licence control and digital rights management routines against malicious reverse engineering. In this thesis, we have taken a complementary approach of addressing the problem of obfuscation by developing weaker notions of security. In this concluding chapter, we first comment on the contributions of the thesis. This is followed by a discussion where we link up the threads of designing intra- and inter-process dependencies. We conclude the thesis with a section on open issues and future work.

8.1 Thesis contributions

The thesis starts by introducing the problem of obfuscation and outlining the drawbacks of alternative software protection techniques. It then lays down the motivation for doing research in obfuscation and for designing practical obfuscating transforms that do not have to depend on cryptographic security. In this section, we summarise the salient contributions of this thesis on a chapter by chapter basis. We omit from this list Chapter 1, which is the introductory chapter, and Chapter 8, which is this concluding chapter of the thesis.

Chapter 2 The primary contribution of this chapter was in outlining the existing definitions of obfuscation and providing a summary of different obfuscating transforms that existed in the literature. It stressed on illustrating the transforms which derived their obfuscatory strength from the hardness of computationally intractable problems. Additionally, it identified the need for evaluating the obfuscatory strength of transforms, which till date is an open problem in software security community.

Chapter 3 The first contribution of this chapter was in describing an attack process that an adversary needs to undertake in order to mount attack on programs obfuscated with resilient transforms. The attack process consisted of four steps and the third step, that of developing an attack model, was identified as the most important step of the attack process. The second contribution of this chapter was in demonstrating that general purpose program analysis tools could be used by an adversary in the attack model. The third contribution here was that of identifying a particular tool, a static slicer, that was used in the attack model for evaluating the obfuscatory strength of all transforms for the rest of the thesis.

Chapter 4 The first contribution of this chapter was in justifying the use of static slicer as a tool in the attack model and developing the notion of "slicing obfuscation". Slicing obfuscations took into account the notion of residues, which are program points that are left behind in the program after slicing. The second contribution

of this chapter was that of deriving metrics based on program residues for quantifying the quality of obfuscating transforms. We called these "residue metrics" and demonstrated their use on a set of small candidate programs. We noted that even with simple slicing obfuscations, the effectiveness of a static slicer as an attack tool could be decreased.

Chapter 5 Proving correctness of obfuscations is a necessary step for guaranteeing that programs give the expected I/O behaviour after they are obfuscated and Drape addressed this issue in his thesis [36]. The main contribution of this chapter was in illustrating the use the Drape's framework for proving correctness of the slicing obfuscations proposed in Chapter 4. This chapter also discussed issues related to placement, localisation, and combination of slicing obfuscations.

Chapter 6 The primary contribution of this chapter was in designing a new form of opaque predicate, called distributed opaque predicate, using the underlying concepts of distributed computing. The chapter illustrated with a running example issues of engineering and embedding distributed opaque predicates in processes executing in distributed computing environment. Additionally, the chapter also described the problem of malicious host in the mobile agents system and illustrated how the use of obfuscated mobile agents could mitigate this issue.

Chapter 7 Chapter 7 deals with evaluation of distributed opaque predicates. The first contribution of this chapter was in adapting the proof rules given by Schlichting and Schneider in [113] for giving proof sketches for programs obfuscated with distributed opaque predicates. The second contribution of this chapter was in evaluating the obfuscatory strength of distributed opaque predicates by adapting the attack process proposed in Chapter 3. It highlighted on the attack model, which comprised of four steps, and argued that while it is easy to attack simple opaque predicates for sequential programs, it is incredibly difficult to attack mobile agents obfuscated with distributed opaque predicates.

8.2 Linking up two approaches

In the thesis we have use two different methodologies for designing obfuscating transforms — used intra-process dependencies for designing slicing obfuscations and inter-process dependencies for designing distributed opaque predicates. The binding thread for both these methodologies is the fact that we used inherent dependencies existing within the program and its execution environment for designing the obfuscations. Also, the attack tool, a static slicer, was used for evaluating the obfuscatory strength of both these methodologies. Our empirical evaluation goal was to see whether an adversary armed with a static slicer will be able to abstract away the parts that are not of interest to him/her. For both the techniques we successfully demonstrated and justified that our obfuscations will make the task of an adversary difficult. It is worth noting that for distributed opaque predicates, our experiments have not been as conclusive as with the experiments conducted for slicing obfuscations. The reason, we believe, is the difficulty of developing tools for successfully slicing message-passing programs. We argued in Chapter 7 that even if such an inter-process static slicer is developed in the future, the adversary will sill not be able to statically determine the runtime values of distributed opaque predicates since global state determination problem of processes executing in message-passing environments is known to be intractable.

8.3 Open issues and future work

A point of contention is the issue of our obfuscations resulting in unmaintainable code if extraneous dependencies and variables are introduced. This could be considered to be a bad software engineering practice. Adding obfuscations to an efficient optimised code is solely the prerogative of a software developer, who may choose to forsake maintainability for the sake of security. We argue that the issue of maintainability is not of much of a concern because distributions of software that are sold commercially can be segregated from the versions that are used for maintenance and in-house development. Thus, “client”

copies can be obfuscated and the "developer" version can be kept in optimised form. A similar issue is whether our obfuscations, especially slicing obfuscations, are vulnerable to compiler optimisation attacks. Indeed, a compiler in its optimisation phase may remove much of the extraneous dependencies but because a software developer is expected to apply obfuscations after the optimisation phase, we do not consider this to be much of a problem.

The most important open issue for our obfuscations is whether their design can be automated without having a human in the loop. The answer to this question is probably a "no" unless we are able to come up with heuristics for taking into account the context sensitive features of a program and generate invariants accordingly. For automating our slicing obfuscations, for example, we need to be able to automatically determine the invariants existing within data structures and constructs of a program before applying the obfuscations. To automate generation of distributed opaque predicates, we need to be able to generate both nondeterministic and deterministic message-passing invariants existing between participating processes. These are non-trivial tasks which we do not expect to be solved in the near future. Although automation is a problem, there are several promising areas in which future work could be pursued. We briefly highlight them below:

For slicing obfuscations

- It will be interesting to see if pointer aliasing could be used as a source for designing the transforms. Since most static analysis tools use conservative analysis, we believe it would be quite easy to include dependencies introduced by pointer aliasing in program slices. The result could be similar to array transformations which were found to be quite redundant since our slicer failed to distinguish array indices — resulting in easy victory against a slicer. If used in the future for other attack tools, we have elegant ways of proving array transformations correct using our imperative proof framework [39].

- Another interesting extension could be the use of inter-procedural dependencies for

generating slicing obfuscations. In languages such as C, function pointers could be used linking up two unrelated methods. It will be interesting to see if our proof framework can handle such constructs.

- We highlighted the issue of composing slicing obfuscations with simple examples and a promising research direction is to see the effect of composed transforms on the residue metrics — it is possible to tune our metrics to obtain a desired degree of obfuscation in the code?

- Placement of transforms and their ordering is also an interesting future research area. Specifically, major advancement can be made in the area of generating heuristics for optimising the placement and order of obfuscations such that they always result in minimum amount of residues.

For our distributed opaque predicates, the following future work can be taken up

- We need more work in the area of generating interesting message-passing invariants. The challenge here is to control the degree of nondeterminism such that a distributed opaque predicate always evaluates to an expected value at predetermined program locations.

- We have not addressed the fault tolerance issues since the objective was to demonstrate how elegantly nondeterminsm of message-passing platforms can be used to design obfuscations. However, when engineering distributed opaque predicates, the processes need to be made aware of the faults that can occur in the middleware and should have routines to handle them. More work needs to be carried out in the area of implementing distributed opaque predicates.

- We considered our predicates to be relational in structure; however, they can also be written in conjunctive and disjunctive forms. It will interesting to see whether the structure of distributed opaque predicates can affect their obfuscatory strength — can we deduce logic to show that predicates of relational structure will be more resilient to attacks than predicates of conjunctive or disjunctive forms?

Bibliography

[1] Business Software Alliance. Second annual BSA and IDC software piracy study, May 2005. Available from URL: www.bsa.org/globalstudy/upload/2005-Global-Study-English.pdf.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, Denmark, May 1994. (DIKU report 94/19).

[3] Paul Anderson and Tim Teitelbaum. Software inspection using CodeSurfer. In *Proceedings of the Workshop on Inspection in Software Engineering (WISE 2001)*, Paris, France, July 2001. IEEE Computer Society.

[4] David Aucsmith. Tamper resistant software: an implementation. *Information Hiding, Lecture Notes in Computer Science*, 1174:317–333, 1996.

[5] Boaz Barak. Can we obfuscate programs? URL: http://www.cs.princeton.edu/~boaz/Papers/obf_informal.html.

[6] Boaz Barak. *Non-Black-Box Techniques in Cryptography*. PhD thesis, Feinberg Graduate School, Weizmann Institute of Science, Israel, 2004.

[7] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18. Springer-Verlag, 2001.

[8] Michael R. Batchelder. Java bytecode obfuscation. Master's thesis, Department of Computer Science, McGill University, Canada, 2007.

[9] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):37–61, 1985.

[10] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2):8, 2007.

[11] David Binkley and Mark Harman. An empirical study of predicate dependence levels and trends. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 330–339, Washington, DC, USA, 2003. IEEE Computer Society.

[12] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, pages 44–53, Washington, DC, USA, 2003. IEEE Computer Society.

[13] David Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computing*, 62:105–178, 2004.

[14] Phillipe Biondi and Fabrice Desclaux. Silver needle in the Skype. Presentation at BlackHat Europe, March 2006. Available from URL: www.blackhat.com/html/bh-media-archives/bh-archives-2006.html.

[15] Alper Caglayan and Colin Harrison. *Agent Sourcebook: A Complete Guide to Desktop, Internet, and Intranet Agents*. Wiley Computer Publishing, 1997.

[16] Mariano Ceccato, Mila Dalla Preda, Jasvir Nagra, Christian Collberg, and Paolo Tonella. Barrier slicing for remote software trusting. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM2007)*, Paris, France, 2007. IEEE Computer Society Press.

[17] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 115–125, New York, NY, USA, 2003. ACM.

[18] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[19] Hoi Chang and Mikhail Atallah. Protecting software code by guards. *Security and Privacy in Digital Rights Management, Lecture Notes in Computer Science*, 2320:160–175, 2002.

[20] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[21] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[22] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *ISC '01: Proceedings of the 4th International Conference on Information Security*, pages 144–155, London, UK, 2001. Springer-Verlag.

[23] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pages 169–186. USENIX Association, August 2003.

[24] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[25] S. Cimato, A. De Santis, and U. Ferraro Petrillo. Overcoming the obfuscation of Java programs by identifier renaming. *Journal of Systems and Software*, 78(1):60–72, 2005.

[26] Joris Claessens, Bart Preneel, and Joos Vandewalle. (How) Can mobile agents do secure electronic transactions on untrusted hosts? A survey of the security issues and the current solutions. *ACM Transactions on Internet Technology (TOIT)*, 3(1):28–48, 2003.

[27] Christian Collberg, Clark D. Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[28] Christian Collberg, Clark D. Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, New York, NY, USA, 1998. ACM Press.

[29] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 167–174, New York, NY, USA, 1991. ACM Press.

[30] J.R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[31] Microsoft Corporation. MSIL.
URL: http://msdn2.microsoft.com/.

[32] Sun Corporation. Java bytecode.
URL: http://java.sun.com/.

[33] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[34] Larry D'Anna, Brian Matt, Andrew Reisse, Tom Van Vleck, Steve Schwab, and Patrick LeBlanc. Self-protecting mobile agents obfuscation report. Technical Report 03-015, Network Associates Laboratories, June 2003. http://www.isso.sparta.com/opensource/jbet/papers/obfreport.pdf.

[35] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[36] Stephen Drape. *Obfuscation of Abstract Data-Types*. DPhil thesis, Oxford University Computing Laboratory, Oxford University, England, 2004.

[37] Stephen Drape. Generalising the array split obfuscation. *Information Sciences*, 177(1):202–219, January 2007.

[38] Stephen Drape, Oege de Moor, and Ganesh Sittampalam. Transforming the .NET Intermediate Language using Path Logic Programming. In *Principles and Practice of Declarative Programming*, pages 133–144. ACM Press, 2002.

[39] Stephen Drape and Anirban Majumdar. Design and Evaluation of Slicing Obfuscations. Technical Report 311, Centre for Discrete Mathematics and Theoretical Computer Science, Department of Computer Science, The University of Auckland, Auckland, New Zealand, June 2007.

[40] Stephen Drape, Anirban Majumdar, and Clark D. Thomborson. Slicing aided design of obfuscating transforms. In *IEEE/ACIS ICIS 2007: In proceedings of the International Computing and Information Systems Conference (ICIS 2007)*, pages 1019–1024, Melbourne, Australia, 2007. IEEE Computer Society.

[41] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM.

[42] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[43] Margaret Ann Francel and Spencer Rugaber. The value of slicing while debugging. *Science of Computer Programming*, 40(2-3):151–169, 2001.

[44] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[45] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[46] Vijay K. Garg. Methods for observing global properties in distributed systems. *IEEE Parallel and Distributed Technology: Systems and Technology*, 5(4):69–77, 1997.

[47] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, 1994.

[48] Programming Languages Research Group. PROLANGS Analysis Framework. URL: http://www.prolangs.rutgers.edu/.

[49] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of theConference on Software Maintenance (CSM 1992)*, pages 299–308, Orlando, FL, November 1992. IEEE Computer Society.

[50] Gael Hachez. *A Comparative Study of Software Protection Tools Suited for E-Commerce with Contributions to Software Watermarking and Smart Cards*. PhD thesis, Universite Catholique de Louvain, Belgium, 2003.

[51] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Notices*, 16(3):63–74, 1981.

[52] Kelly Heffner and Christian Collberg. The obfuscation executive. In *ISC 2004: 7th International Information Security Conference*, pages 428–440, New York, NY, USA, 2004. Lecture Notes in Computer Science, Vol. 3225.

[53] Michael Hind. Npic — new paltz interprocedural compiler. *SIGSOFT Softw. Eng. Notes*, 25(1):57–58, 2000.

[54] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999.

[55] Tommy Hoffner, Mariam Kamkar, and Peter Fritzson. Evaluation of program slicing tools. In *Automated and Algorithmic Debugging*, pages 51–69, 1995.

[56] Fritz Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Mobile Agents and Security*, pages 92–113, London, UK, 1998. Springer-Verlag.

[57] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert Tarjan. Dynamic self-checking techniques for improved tamper resistance. *Security and Privacy in Digital Rights Management, Lecture Notes in Computer Science*, 2320:141–159, 2002.

[58] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 1990. ACM Press.

[59] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):1–6, 1997.

[60] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[61] PreEmptive Solutions Inc. Dash-O.
URL: www.preemptive.com/products/dasho.

[62] PreEmptive Solutions Inc. Dotfuscator.
URL: www.preemptive.com/products/dotfuscator.

[63] Kirill S. Ivanov and Vladimir A. Zakharov. Program obfuscation as obstruction of program static analysis. In *Volume 6. ISPRAN 2005. Russian Academy of Sciences Technical Report Series*, pages 137–156. ISPRAN 2005, 2005.

[64] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 329–341, New York, NY, USA, 1998. ACM.

[65] Ganeshan Jayaraman, Venkatesh Prasad Ranganath, and John Hatcliff. Kaveri: Delivering the Indus Java program slicer to Eclipse. In *FASE*, pages 269–272. Lecture Notes In Computer Science, SpringerVerlag, 2005.

[66] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag.

[67] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[68] Dieter Kranzlmüller and Martin Schulz. Notes on nondeterminism in message passing programs. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 357–367, London, UK, 2002. Springer-Verlag.

[69] Argonne National Laboratory. MPI: Message Passing Interface.
URL: http://www-unix.mcs.anl.gov/mpi/.

[70] Oak Ridge National Laboratory. PVM: Parallel Virtual Machine.
URL: http://www.csm.ornl.gov/pvm/.

[71] Telecom Italia Labs. JADE — Java Agent DEvelopment framework.
URL: http://jade.tilab.com/.

[72] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[73] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[74] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, Department of Computer Science, McGill University, Canada, 2002.

[75] Hon F. Li, Juergen Rilling, and Dhrubajyoti Goswami. Granularity-driven dynamic predicate slicing algorithms for message passing systems. *Automated Software Engineering*, 11(1):63–89, 2004.

[76] Jiming Liu. *Autonomous agents and multi-agent systems: explorations in learning, self-organization and adaptive computation.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2001.

[77] Douglas Low. Java control flow obfuscation. Master's thesis, Department of Computer Science, The University of Auckland, New Zealand, 1998.

[78] James Robert Lyle. *Evaluating variations on program slicing for debugging.* PhD thesis, University of Maryland, College Park, USA, 1984.

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

[80] Matias Madou, Ludo Van Put, and Koen De Bosschere. Understanding obfuscated code. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 268–274, Washington, DC, USA, 2006. IEEE Computer Society.

[81] Anirban Majumdar, Stephen Drape, and Clark D. Thomborson. Metrics-based evaluation of obfuscating transforms. In *IAS 2007: Proceedings of the Third International Symposium on Information Assurance and Security (IAS 2007)*, Manchester, United Kingdom, 2007. IEEE Computer Society.

[82] Anirban Majumdar, Antoine Monsifrot, and Clark D. Thomborson. On evaluating obfuscatory strength of alias-based transforms using static analysis. In *ADCOM 2006: Proceedings of the 14th International Conference on Advanced Computing and Communication (ADCOM 2006)*, Mangalore, India, 2006. IEEE Computer Society.

[83] Anirban Majumdar and Clark D. Thomborson. Securing mobile agents control flow using opaque predicates. In *9th International Conference on Knowledge-based Intelligent Information and Engineering Systems, (KES 2005)*, Melbourne, Australia, 2005. Springer-Verlag.

[84] Anirban Majumdar and Clark D. Thomborson. Interpreting opacity in the context of information-hiding and obfuscation in distributed systems. In *TENCON 2006. 2006 IEEE Region 10 Technical Conference*, pages 1–4, Hong Kong, 2006. IEEE Computer Society.

[85] Anirban Majumdar and Clark D. Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In *ACSC '06: Proceedings of the 29th Australasian Computer Science Conference*, pages 187–196, Hobart, Australia, 2006. ACM Press and Australian Computer Society.

[86] Anirban Majumdar, Trong Khiem Tran, Eslam Al Maghayreh, Hon Fung Li, and Dhrubajyoti Goswami. On-the-fly agent-based distributed shared state monitoring. In *PDPTA*, pages 1396–1402, Las Vegas, NV, USA, 2004. CSREA Press.

[87] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[88] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[89] Timothy M. Meyers and David Binkley. Slice-based cohesion metrics and software intervention. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 256–265, Washington, DC, USA, 2004. IEEE Computer Society.

[90] Neeraj Mittal and Vijay K. Garg. On detecting global predicates in distributed computations. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 3–10, Washington, DC, USA, 2001. IEEE Computer Society.

[91] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[92] John C. Munson and Taghi M. Khoshgoftaar. Measurement of data structure complexity. *Journal of Systems and Software*, 20(3):217–225, 1993.

[93] Jasvir Nagra. *Threading Software Watermarks*. PhD thesis, Department of Computer Science, The University of Auckland, 2006.

[94] Masahide Nakamura, Akito Monden, Tomoaki Itoh, Ken ichi Matsumoto, Yuichiro Kanzaki, and Hirotsugu Satoh. Queue-based cost evaluation of mental simulation process in program comprehension. In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, page 351, Washington, DC, USA, 2003. IEEE Computer Society.

[95] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, pages 61–91, London, UK, 1998. Springer-Verlag.

[96] University of Arizona. SandMark: A tool for the study of software protection algorithms. URL: http://sandmark.cs.arizona.edu.

[97] University of California-Berkeley. SETI@home. URL: http://setiathome.berkeley.edu/.

[98] University of Ghent. Diablo: A retargetable link-time binary rewriting framework. URL: http://diablo.elis.ugent.be.

[99] Toshio Ogiso, Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEICE Transactions on Fundamentals*, E86-A(1):1–11, 2003.

[100] Linda M. Ott and Jeffrey J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Software Metrics Symposium*, pages 78–81, 1993.

[101] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM Press.

[102] Özalp Babaoğlu and Keith Marzullo. *Distributed systems (2nd Ed.)*, chapter Consistent global states of distributed systems: fundamental concepts and mechanisms, pages 55–96. ACM Press/Addison-Wesley Publishing Co., 1994.

[103] Jens Palsberg, Sowmya Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Yi Zhang. Experience with software watermarking. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, pages 308–317, Washington, DC, USA, 2000. IEEE Computer Society.

[104] Tao Pang. *An Introduction to Computational Physics*. Cambridge University Press, 1997. URL: www.physics.unlv.edu/~pang/cp.html.

[105] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(1):4, 2007.

[106] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388, New York, NY, USA, 2007. ACM Press.

[107] ProGuard. Java obfuscator.
URL: http://proguard.sourceforge.net/.

[108] Ganeshan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[109] Juergen Rilling and Tuomas Klemola. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 115–124, Washington, DC, USA, 2003. IEEE Computer Society.

[110] Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Java obfuscation with a theoretical basis for building secure mobile agents. In Antonio Lioy and Daniele Mazzocchi, editors, *Communications and Multimedia Security*, volume 2828 of *Lecture Notes in Computer Science*. Springer, 2003.

[111] Tomas Sander and Christian F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, pages 44–60, London, UK, 1998. Springer-Verlag.

[112] Nuno Santos, Pedro Pereira, and Luís Moura e Silva. A Generic DRM Framework for J2ME Applications. In Olli Pitkänen, editor, *First International Mobile IPR Workshop: Rights Management of Information (MobileIPR)*, pages 53–66. Helsinki Institute for Information Tecnhology, August 2003.

[113] Richard D. Schlichting and Fred B. Schneider. Using message passing for distributed programming: proof rules and disciplines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(3):402–431, 1984.

[114] Robert R. Schneck and George C. Necula. A gradual approach to a more trustworthy, yet scalable, proof-carrying code. In *CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, pages 47–62, London, UK, 2002. Springer-Verlag.

[115] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):540–582, 2000.

[116] Mikhail Sosonkin, Gleb Naumovich, and Nasir Memon. Obfuscation of design intent in object-oriented applications. In *DRM '03: Proceedings of the 3rd ACM workshop on Digital rights management*, pages 142–153, New York, NY, USA, 2003. ACM Press.

[117] Mikhail Sosonkin, Hong Heather Yu, Nasir D. Memon, Ezgi Yalcin, and Gleb Naumovich. *Class coalescence for obfuscation of object-oriented software.* United States Patent 20040103404A1, 2006. Assignee: Matsushita Electric industrial Co., Ltd.

[118] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[119] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 315–330, New York, NY, USA, 2004. ACM Press.

[120] Mirko Streckenbach and Gregor Snelting. Points-to for Java: A general framework and an empirical comparison. Unpublished report. Department of Computer Science, University of Passau, Germany. 10pp, Obtained through email correspondence in March 2005.

[121] Katia Sycara, Anandee Pannu, Mike Williamson, Dajun Zeng, and Keith Decker. Distributed intelligent agents. *IEEE Expert: Intelligent Systems and Their Applications*, 11(6):36–46, 1996.

[122] Clark D. Thomborson, Jasvir Nagra, Ram Somaraju, and Charles He. Tamper-proofing software watermarks. In *ACSW Frontiers '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 27–36, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[123] Frank Tip. A survey of program slicing techniques. Technical Report CS-R9438, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1994.

[124] Paul M. Tyma. *Method for renaming identifiers of a computer program.* United States Patent No. 6102966, 2000. Assignee: PreEmptive Solutions, Inc.

[125] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 45–54, Washington, DC, USA, 2005. IEEE Computer Society.

[126] Kansas State University. Indus Java Slicer.
URL: http://indus.projects.cis.ksu.edu/.

[127] McGill University. Soot: a Java optimization framework.
URL: http://www.sable.mcgill.ca/soot/.

[128] Stanford University. Joeq compiler system.
URL: http://joeq.sourceforge.net/.

[129] University of Texas at Austin CS1713 Course.
URL: www.cs.utexas.edu/users/djimenez/utsa/cs1713-3/c/.

[130] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 18–34, London, UK, 2000. Springer-Verlag.

[131] Ashok P. Ramasamy Venkatraj. Program obfuscation. Master's thesis, Department of Computer Science, University of Arizona, USA, 2003.

[132] Chenxi Wang. *A Security Architecture for Survivability Mechanisms*. PhD thesis, Department of Computer Science, University of Virginia, USA, 2000.

[133] Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. Protection of software-based survivability mechanisms. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 193–202, Washington, DC, USA, 2001. IEEE Computer Society.

[134] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[135] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206, New York, NY, USA, 1999. ACM Press.

[136] Gregory Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Institute of Engineering Cybernetics, Wroclaw University of Technology, Poland, 2002.

[137] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[138] Zelix. Klassmaster.
URL: http://www.zelix.com/klassmaster/.

[139] Xiangyu Zhang and Rajiv Gupta. Hiding program slices for software security. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 325–336, Washington, DC, USA, 2003. IEEE Computer Society.

Publications from thesis

Chapter 2

- Majumdar, A., Thomborson, C. D., and Drape, S. J.: A Survey of Control-Flow obfuscations, in proceedings of the *2nd International Conference on Information Systems Security (ICISS 2007)*. December 17-21, 2006. Calcutta, India. Springer-Verlag.

Chapter 3

- Majumdar, A., Monsifrot, A., and Thomborson, C. D.: On Evaluating Obfuscatory Strength of Alias-based Transforms Using Static Analysis, in proceedings of the *14th International Conference on Advanced Computing and Communication (ADCOM 2006)*. December 20-23, 2006. Mangalore, India. IEEE Computer Society.

Chapters 4 and 5

- Drape, S. J., Majumdar, A., and Thomborson, C. D.: Slicing Aided Design Of Obfuscating Transforms, in proceedings of the *IEEE/ACIS Conference (ICIS 2007)*. July 11-13, 2007. Melbourne, Australia. IEEE Computer Society.

- Drape, S. J. and Majumdar, A.: Design and Evaluation of Slicing Obfuscations, *Centre for Discrete Mathematics and Theoretical Computer Science (CDMTCS) Technical Report # 311*, Department of Computer Science, University of Auckland.

- Drape, S. J., Thomborson, C. D., and Majumdar. A.: Specifying Imperative Data Obfuscations, to appear in proceedings of the *10th Information Security Conference (ISC 2007)*. October 9-12, 2007. Valparaiso, Chile. Springer-Verlag.

- Majumdar, A., Drape, S. J., and Thomborson, C. D.: Metrics-based Evaluation of Slicing Obfuscations, in proceedings of the *3rd International Symposium on Information Assurance and Security (IAS 2007)*. August 29-31, 2007. Manchester, United Kingdom. IEEE Computer Society.

- Majumdar, A., Drape, S. J., and Thomborson, C. D.: Slicing Obfuscations: Design, Correctness, and Evaluation, to appear in proceedings of the *7th ACM Workshop on Digital Rights Management (ACM-DRM 2007)*. October 29 - November 2, 2007. Alexandria, VA, USA. ACM Press.

Chapters 6 and 7

- Majumdar, A. and Thomborson, C. D.: Securing Mobile Agents Control Flow Using Opaque Predicates, in proceedings of *Workshop on Intelligent Information Hiding and Multimedia Signal Processing (IIHMSP 2005)*. September 14-16, 2005. Melbourne, VIC, Australia. Springer-Verlag.

- Majumdar, A. and Thomborson, C. D.: On the Use of Opaque Predicates in Mobile Agent Code Obfuscation (Extended Abstract), in proceedings of the *IEEE Conference on Intelligence and Security Informatics (ISI 2005)*. May 15-19, 2005. Atlanta, GA, USA. IEEE Computer Society.

- Majumdar, A. and Thomborson, C. D.: Interpreting Opacity in the Context of Information-hiding and Obfuscation in Distributed Systems, in proceedings of the *IEEE TENCON*. November 14-17, 2006. Hong Kong. IEEE Computer Society.

- Majumdar, A. and Thomborson, C. D.: Manufacturing Opaque Predicates in Distributed Systems for Code Obfuscation, in proceedings of the *29th Australasian Computer Science Conference (ACSC 2006)*. January 15-19, 2006. Hobart, TAS, Australia. ACM International Conference Proceeding Series; Vol. 171., ACM Press.