Program Code Analysis and Manipulation

Ph.D. Dissertation

by

Ákos Kiss

Supervisor: Dr. Tibor Gyimóthy

Submitted to the

Ph.D. School in Mathematics and Computer Science

Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged, 2008

Foreword

Eight years. This is how long ago I started as a young researcher, and now, after all these years, I am about to finish my PhD thesis. Eight years is a long time. This is a quarter of my life so far. Most of this time has been devoted to research: confronting interesting problems, looking for and experimenting with novel approaches, and finding the best solutions. This thesis tries to summarise the results of this eight-year research period.

These years have been quite result-oriented. However, I have recently realised that they should have been a little more people-oriented, perhaps. Many people were around me who contributed to the writing of this thesis – either professionally or personally – and often I forgot to say at least thank-you to them. Thus, at this very beginning of my thesis, I would like to thank the continuous support of my family, my colleagues, and my friends. Without them, I would not have been able to write this work.

Firstly, I would like to thank Tibor Gyimóthy, my supervisor, for directing me as a young researcher, and always offering interesting and challenging problems. Secondly, to Dave Binkley, Sebastian Danicic, and Mark Harman, whom I consider both mentors and friends. Thirdly, my thanks goes to all my colleagues and co-authors, namely Árpád Beszédes, David Curley, Adrienne Dobóczky, Rudolf Ferenc, Patrícia Frittman, Zoltán Herczeg, Judit Jász, Bogdan Korel, Tímea László, Gábor Lehotai, Lahcen Ouarbya, István Siket, and Péter Siket. In one way or another, they all have contributed to the creation of this thesis. Finally, I would like to express my gratitude for the continuous support of my mother, my father, and my wife, Kinga.

Ákos Kiss, September 2008.

Contents

List of Figures

List of Tables

To Máté and Dávid

Chapter 1

Preface

"While much attention in the wider software engineering community is properly directed towards other aspects of systems development and evolution, such as specification, design and requirements engineering, it is the source code that contains the only precise description of the behaviour of the system. The analysis and manipulation of source code thus remains a pressing concern."

The above sentences constitute the motto of SCAM, the annual conference on Source Code Analysis and Manipulation, and this is what motivated the author while doing his research work. The field of code analysis and manipulation is huge; it includes topics like program transformation, abstract interpretation, program slicing, source level software metrics, decompilation, source level testing and verification, source level optimisation and program comprehension among others. Out of these numerous topics, the author focused on three issues: the theoretical foundation of program slicing, the application of program slicing to binary programs, and the obfuscation of programs written in C++ language.

The structure of the main body of the thesis follows the previous list of research topics. Part I is devoted to the theory of program slicing, Part II is about the slicing of binary programs, while Part III discusses the application of code obfuscation to C++ source code. Each part of the thesis begins with an introduction to and motivation for the specific area, and then following

the presentation of the results a there is a summary. After the main parts, the appendices follow with the summary of the whole thesis (both in English and in Hungarian), a description of the author's contributions to the key results, a description of the related work and bibliographic references.

The author admits that Part II, i.e., the slicing of binaries, might seem inappropriate in the context of source code analysis. However, for the scientific community of SCAM, 'source code' is any fully executable description of a software system. Thus, this definition not only covers high level languages but includes machine code as well. Even though this relaxed definition nicely incorporates all three main parts of this work, the thesis has been titled *Program Code Analysis and Manipulation* to match the terminology used by the wider software engineering community.

In addition, the author remarks that although the results presented in this thesis are his major contribution, from this point on, the term *'we'* will be used instead of *'I'* for self reference to acknowledge the contribution of the co-authors of the papers that this thesis is based on.

Part I

The Theory of Slicing

Chapter 2

Introduction to Slicing and its Theory

Program slicing is a technique for extracting the parts of a program which affect a given set of variables of interest, and was originally introduced by Mark Weiser in 1979 [102]. By focusing on the computation of only a few variables, the slicing process can be used to eliminate the parts of the program which cannot affect these variables. This way the size of the program is reduced. The reduced program is called a slice.

Slicing has many applications because it allows a program to be simplified by focusing attention on a sub–computation of interest for a chosen program. The user specifies the sub–computation of interest using a 'slicing criterion'. This first part of the thesis is concerned with the relationships between the slicing criteria for the dynamic and the static forms of slicing and the sets of slices allowable according to the different slicing techniques which use these criteria.

Among other applications, slicing has been applied in reverse engineering [23, 95], program comprehension [36, 52], software maintenance [22, 29, 43, 42], debugging [2, 61, 80, 105], testing [16, 48, 50, 56, 57], component re–use [7, 28], program integration [19, 58], and software metrics [10, 71, 86]. In the literature there are several surveys on slicing techniques, applications and variations [17, 18, 35, 53, 98].

Slices can be constructed statically [104, 60] or dynamically [66, 3]. In static slicing, the input to the program is unknown and the slice must therefore preserve the meaning for all possible inputs. By contrast, in dynamic slicing, the input to the program is known, and so the slice only needs to preserve the meaning for the input under consideration. Dynamic slicing is especially useful in applications like debugging, where the input to the program has a crucial bearing on the problem in hand.

Here, we are interested in the formal definitions and properties of slicing (rather than in algorithms for computing them). We shall employ the projection theory of program slicing introduced by Harman, Danicic, and Binkley [49, 51], which was first used to examine the similarities and differences between the amorphous and the syntax-preserving forms of slicing. This study uses projection theory to investigate the nature of dynamic slicing, as originally formulated by Korel and Laski [66].

The next few chapters will make abundantly clear that the dynamic slicing criterion is more subtle than previous authors have observed [21, 41, 100]. There is no simple two-element subsumption relationship between Korel and Laski dynamic slicing and static slicing. Previous authors regarded the addition of program input as the only aspect separating the static and the dynamic slicing criteria. However, projection theory allows for the analysis of the 'subsumption' relationship between various formulations of static and dynamic slicing, and the analysis reveals the existence of new, as yet unexplored slicing criteria which may find applications in their own right. This part of the thesis also proves that the 'subsumption' relationship for the semantic properties of slicing criteria is respected by all the definitions of slicing which use the standard statement deletion.

However, subsumption is not the only interesting relationship between slicing techniques. In any application of slicing, the size of the slices is crucial: the smaller, the better. Thus, we want to put statements such as "dynamic slices are smaller than static slices" on a firm theoretical footing. We intuitively know what we mean by such statements, but capturing this formally is non-trivial. Needless to say, not all dynamic slices are smaller than all static slices, since there is the complication of which particular dynamic

slicing definition one is to adopt; some are incomparable with static slicing. Therefore, in the following we will define a relationship that allows us to determine whether one definition of slicing leads to inherently smaller slices than another.

Chapter 3

Background

3.1 The Program Projection Theory

Program Projection Theory [49, 51] is, in essence, a generalisation of slicing. It uses two relations over programs: a *pre-order*, i.e., a transitive and reflexive relation called *syntactic ordering*, and an *equivalence relation* called *semantic equivalence*. Syntactic ordering is used to capture the syntactic property that slicing seeks to optimise. Programs that are lower according to the ordering are considered to be 'better'. The semantic relation is an equivalence relation which captures the semantic property that remains invariant during slicing.

Definition 3.1 (Syntactic Ordering)**.** A syntactic ordering, denoted by $\lesssim$, is a computable transitive and reflexive relation on programs.

Definition 3.2 (Semantic Equivalence)**.** A semantic equivalence, denoted by $\approx$, is an equivalence relation on program semantics.

Definition 3.3 ($(\lesssim, \approx)$ Projection)**.** Given syntactic ordering $\lesssim$ and semantic equivalence $\approx$,

$$\begin{gathered}\text{program } q \text{ is a } (\lesssim, \approx) \textit{ projection} \text{ of program } p \\ \text{if and only if} \\ q \lesssim p \ \wedge \ q \approx p.\end{gathered}$$

That is, in a projection, the syntax can only improve while the semantics of interest must remain unchanged. Projection theory, thus, elegantly separates the syntactic and semantic constraints inherent in program slicing.

3.2 The Syntactic Ordering Induced by Statement Deletion

The following definition formalises the oft-quoted remark: "a slice is a subset of the program from which it is constructed". It defines the syntactic ordering for syntax-preserving slicing. Note that for ease of presentation, it is assumed that each program component occupies a unique line. Thus, a line number can be used to uniquely identify a particular program component. In this thesis, just the syntax-preserving forms of slicing are considered [49], so all the following slicing definitions will share the following Syntactic Ordering.

Definition 3.4 (Traditional Syntactic Ordering)**.** Let F be a function that takes a program and returns a partial function from line numbers to statements, such that the function $F(p)$ maps l to c if and only if program p contains the statement c at line number l. Traditional syntactic ordering, denoted by $\sqsubseteq$, is defined as follows:

$$p \sqsubseteq q \Leftrightarrow F(p) \subseteq F(q).$$

3.3 Weiser's Static Backward Slicing

The semantic property that static slicing respects is based upon the concept of *state trajectory*. According to Weiser, informally, for program q to be a static slice of program p with respect to slicing criterion (V, n), the trajectories of p and q must semantically agree with respect to variables V at line n for every initial state. This means that if we remove all elements from the state trajectories apart from those which mention n and then, in what remains, just consider the subset of the states which are concerned with the set of variables V, the two trajectories should appear to be identical. There

is, of course, also the syntactic requirement that q must be obtained from p by statement deletion, i.e. $q \sqsubseteq p$.

The following definitions of *state trajectory*, *state restriction*, $Proj$, and $Proj'$ come from Weiser's definition of slice semantics [104].

Definition 3.5 (State Trajectory)**.** A *state trajectory* is a finite sequence of (line number, state) pairs:

$$(l_1, \sigma_1)(l_2, \sigma_2) \ldots (l_k, \sigma_k),$$

where a state is a partial function mapping a variable to a value, and entry i is (l_i, σ_i) if after i statement executions the state is σ_i, and the next statement to be executed is at line number l_i.

This definition considers only terminating programs. Both Weiser and subsequent authors remain silent on the required behaviour of a slice in situations where the original program fails to terminate. In this thesis, it is Weiser's definition of slicing which will be adopted. The definitions presented give rise to semantic equivalence relations over terminating programs. Like previous authors, the present definitions do not define the meaning of a slice for programs which fail to terminate.

Definition 3.6 (State Restriction)**.** Given a state, σ and a set of variables V, $\sigma \restriction V$ restricts σ so that it is defined only for variables in V:

$$(\sigma \restriction V)x = \begin{cases} \sigma\, x & \text{if } x \in V, \text{ and} \\ \bot & \text{otherwise.} \end{cases}$$

Definition 3.7 ($Proj$)**.** For slicing criterion (V, n), and state trajectory $T = (l_1, \sigma_1)(l_2, \sigma_2) \ldots (l_k, \sigma_k)$,

$$Proj_{(V,n)}(T) = Proj'_{(V,n)}(l_1, \sigma_1) \ldots Proj'_{(V,n)}(l_k, \sigma_k),$$

where

$$Proj'_{(V,n)}(l, \sigma) = \begin{cases} (l, \sigma \restriction V) & \text{if } l = n, \text{ and} \\ \lambda & \text{otherwise,} \end{cases}$$

1	`n=input();`	1	`n=input();`
2	`s=0;`		
3	`p=1;`	3	`p=1;`
4	`while (n>1) {`	4	`while (n>1) {`
5	`  s=s+n;`		
6	`  p=p*n;`	6	`  p=p*n;`
7	`  n=n-1;`	7	`  n=n-1;`
8	`}`	8	`}`
9	`output(s);`		
10	`output(p);`	10	`output(p);`
$p_{3.1}$: Original program		$q_{3.1}$: Slice w.r.t. $(\{\texttt{p}\}, 10)$	

Figure 3.1: A program and one of its static slices.

where λ denotes the empty string.

Having defined the necessary auxiliary functions, we are now in a position to define static backward equivalence, the semantic relationship preserved by backward static slicing, as originally defined by Weiser [102].

Definition 3.8 (Static Backward Equivalence)**.** Given two programs p and q, and slicing criterion (V, n), p is *static backward equivalent* to q, written $p\ \mathcal{S}(V, n)\ q$, if and only if for all initial states σ, when the execution of p in σ gives rise to a state trajectory T_p^σ and the execution of q in σ gives rise to a state trajectory T_q^σ, then $Proj_{(V,n)}(T_p^\sigma) = Proj_{(V,n)}(T_q^\sigma)$.

The static slicing semantic equivalence relation is parameterised by V and n, and hence it really defines a function from slicing criterion (V, n), to equivalence relations over programs. This reflects the fact that each slicing criterion yields slices that respect a different projection of the semantics of the program from which they are constructed. Instantiating Definitions 3.4 and 3.8 into Definition 3.3, yields the following:

Definition 3.9 (Static Backward Slicing)**.** A program q is a *static backward slice* of a program p with respect to the slicing criterion (V, n) if and only if q is a $(\sqsubseteq, \mathcal{S}(V, n))$ projection of p.

As an example, consider programs $p_{3.1}$ and $q_{3.1}$ in Figure 3.1. Program $q_{3.1}$ is a static backward slice of $p_{3.1}$, because $p_{3.1}$ is static backward equivalent

to $q_{3.1}$, written $p_{3.1}\ \mathcal{S}(\{\mathtt{p}\}, 10)\ q_{3.1}$, since for every initial state σ,

$$Proj_{(\{\mathtt{p}\},10)}(T^{\sigma}_{p_{3.1}}) = Proj_{(\{\mathtt{p}\},10)}(T^{\sigma}_{q_{3.1}})$$

and $q_{3.1}$ is obtained from $p_{3.1}$ by statement deletion.

3.4 The Dynamic Slicing of Korel and Laski

Static slices must preserve a projection of the semantics of the original program for *all* possible program inputs. In certain applications this requirement is too strict. For example, when debugging, only a single input is often of interest. Korel and Laski [66] were the first to introduce a dynamic definition of a slice. A dynamic slice only needs to preserve the effect of the original program upon the slicing criterion for a single input. The dynamic paradigm is ideally suited to problems such as bug-location, because a bug is typically detected as the result of the execution of a program with respect to some specific inputs.

Consider once again the example in Figure 3.1, but with `p=1` mistakenly coded as `p=0`. Suppose the original program is executed and given the input 1. The value of `p` at the end of the execution is incorrect — it is 0 when it should be 1. The dynamic slice identifies those statements that contribute to the value of the variable `p` when the input 1 is supplied to the program; in this case, just the line `p=0`. Locating the bug (the faulty initialisation of `p`) in terms of the dynamic slice is thus easier than with either the original program or the corresponding static slice.

This is a rather contrived example as the input causes the `while` loop to go un-executed. However, in general, dynamic slicing improves precision in several ways. One is that statements which remain un-executed are not included in a dynamic slice. Another is that statements which are executed and create data and control dependencies may be removed from the slice, should these dependencies be subsequently 'overwritten' during the execution. Yet another is that dynamic slicing has more precise information concerning the value of array indices and pointer variables, which allows a more precise

determination of data dependencies.

The literature on dynamic slicing includes many different algorithms [3, 9, 46, 62, 66, 68]. Many of these algorithms do not necessarily output executable programs [3, 9, 46, 62]. Rather, they regard a dynamic slice as the collection of statements that have an effect upon the slicing criterion given the chosen input. By contrast, this part of the thesis is concerned solely with the executable forms of slicing.

As defined by Korel and Laski, an executable dynamic slicing criterion is (x, I^q, V), which like the static slicing criterion (V, n) includes a set of variables V. Unlike the static slicing criterion, it also includes the program's input x and replaces the location of interest n with I^q, which is the qth instruction in the execution trajectory, which is I. Thus, a slice can be taken with respect to a particular instance rather than all the instances of a statement (instruction) from the program.

The definition uses two auxiliary functions on sequences, $Front$ and DEL [66]. $Front(T, i)$ is the 'front' i elements of sequence T from 1 to i inclusive. $DEL(T, \pi)$ is a filtering operation which takes a predicate π and returns the sequence obtained by deleting the elements of T that satisfy π.

The following definition is taken verbatim from Korel and Laski's work on dynamic slicing [66], even if this causes some inconsistencies in the notations used in the thesis. E.g., in Korel and Laski's interpretation, a trajectory is simply a finite sequence of line numbers, as opposed to Weiser's definition of a *state* trajectory (see Definition 3.5). However, in the following, we will refer to both kinds of trajectories as a *trajectory* and the context will make clear which meaning is involved.

Definition 3.10 (Korel and Laski's Dynamic Slice)**.** Let $c = (x, I^q, V)$ be a slicing criterion of a program p and T the trajectory of p on input x. A *dynamic slice* of p on c is any executable program p' that is obtained from p by deleting zero or more statements such that when executed on input x, produces a trajectory T' for which there exists an execution position q' such that

(KL1) $Front(T', q') = DEL(Front(T, q),\ T(i)\ \notin\ N'\ \wedge\ 1\ \leq\ i\ \leq\ q)$,

(KL2) for all $v \in V$, the value of v before the execution of instruction $T(q)$ in T equals the value of v before the execution of instruction $T'(q')$ in T',

(KL3) $T'(q') = T(q) = I$,

where N' is a set of instructions in p'.

It would be nice to define a projection corresponding precisely to the dynamic slice defined by Korel and Laski. However, as it will be seen, this requires quite some effort. Thus, inserting dynamic slicing into the framework of projection theory will be left for the next chapter.

Chapter 4

The Unified Framework

4.1 Comparison of Weiser's Static Slicing with Korel and Laski's Dynamic Slicing

A common view is that every static slice is (an overly large) Korel-and-Laski-style dynamic slice as well. One intuitively expects that a dynamic slicing criterion is looser than a static one, since it preserves the semantics of a program for only one fixed input instead of all the possible ones. Moreover, a dynamic slicing criterion selects just one occurrence of an instruction from the trajectory, as opposed to static slicing where all occurrences of the point of interest are taken into account.

However, as Figure 4.1 shows, Korel and Laski's (KL) definition of dynamic slicing is incomparable with the definition of static slicing, since not

1 x=1; 2 x=2; 3 if (x>1) 4 y=1; 5 else 6 y=1; 7 z=y;	1 x=1; 3 if (x>1) 4 y=1; 5 else 6 y=1; 7 z=y;
$p_{4.1}$: Original Program	$q_{4.1}$: Slice w.r.t. $(\{\mathtt{y}\}, 7)$

Figure 4.1: A static slice, which is not a KL–slice.

all static slices are appropriate KL–slices (and not all KL–slices are static slices, which is trivial). In Figure 4.1, program $q_{4.1}$ is a valid static slice with respect to $(\{\texttt{y}\}, 7)$ since at Line 7 the value of `y` is 1 for all inputs, just like in $p_{4.1}$. However, $q_{4.1}$ is not a Korel-and-Laski-style dynamic slice of $p_{4.1}$ with respect to $(\langle\rangle, 7^4, \{\texttt{y}\})$, because the trajectory of $p_{4.1}$ is (2 3 4 7), but the trajectory of $q_{4.1}$ is (3 6 7). Having different execution paths violates KL1 (see Definition 3.10), since the truncated and filtered trajectories differ, i.e., (3 4 7) $\neq$ (3 6 7).

Notice that the cause of incomparability between KL–dynamic-slicing and static slicing is that KL–dynamic-slicing is "looser" as it must preserve behaviour for just a single input (a desired effect) while, because of KL1, it is also more strict. Thus, restriction KL1 can prevent us from choosing an otherwise acceptable program from several semantically equivalent programs.

4.2 The Unified Equivalence

Now that the main cause of imcomparability between Weiser's static slicing and KL–slicing has been identified, KL–slicing can be incorporated into the framework of projection theory. However, Definition 3.7 is not sufficient for this purpose as it cannot capture the execution path (KL1) requirement.

To set up a unified framework we shall extend these definitions by introducing counterparts to $Proj$ and $Proj'$ named $Proj^*$ and $Proj'^*$, respectively. The extension splits the "statement" parameter n into P and I: P, an instruction-natural number pair, identifies those instruction occurrences from the trajectory whose semantics must be preserved. Parameter I captures the trajectory requirement of KL1 by keeping only the line number, in the form of $(n, \perp)$, for those instructions that are not in the slicing criterion but get executed.

Notice that in the following definitions the notation is different from the one used by Korel and Laski. While, in Definition 3.10, I^q represents the qth instruction in the trajectory, which is I, $n^{(k)}$ is used to denote the kth occurrence of instruction n in the trajectory and the exact position is only implicitly given. This difference makes it possible to capture the iteration

count component of the Korel and Laski slicing criterion.

Definition 4.1 ($Proj'^{*}$)**.** $Proj'^{*}$ is defined in terms of 5 parameters: a set of variables V, a set of (line number, natural number) pairs P, a set of line numbers I, a (line number, natural number) pair $n^{(k)}$, and a state σ:

$$Proj'^{*}_{(V,P,I)}(n^{(k)}, \sigma) = \begin{cases} (n, \sigma \restriction V) & \text{if } n^{(k)} \in P, \\ (n, \bot) & \text{if } n^{(k)} \notin P \text{ and } n \in I, \\ \lambda & \text{otherwise.} \end{cases}$$

Note that $Proj'^{*}_{(V,P,I)}(n^{(k)}, \sigma)$ evaluates either to a single pair or an empty sequence of pairs depending on its parameters. It, in effect, keeps a projected version of each pair if either $n^{(k)} \in P$ or $n \in I$ otherwise it 'throws away' the pair completely.

Definition 4.2 ($Proj^{*}$)**.** For a set of variables V, set of (line number, natural number) pairs P, set of line numbers I and state trajectory T:

$$Proj^{*}_{(V,P,I)}(T) = Proj'^{*}_{(V,P,I)}({n_1}^{(k_1)}, \sigma_1) \ldots Proj'^{*}_{(V,P,I)}({n_l}^{(k_l)}, \sigma_l),$$

where k_i is the number of occurrences of n_i in the first i elements of T (i.e., ${n_i}^{(k_i)}$ is the most recent occurrence of n_i in $T(1) \ldots T(i)$), and l is the highest index in T such that ${n_l}^{(k_l)} \in P$.

Observe that if $P = \{n\} \times \mathbf{N}$, where $\mathbf{N}$ is the set of natural numbers, and $I = \emptyset$ then $Proj^{*}_{(V,P,I)}(T) = Proj_{(V,n)}(T)$, since the middle case of $Proj'^{*}$ can be dropped. This leaves Weiser's definition of $Proj$. However, by choosing different values for P and I, $Proj^{*}$ can capture Korel and Laski's requirements as well. Consider again the program $p_{4.1}$ from Figure 4.1. If $V = \{\mathtt{y}\}$, $P = \{7^{(1)}\}$, and $I = \{1, 3, 4, 5, 6, 7\}$ then $Proj^{*}_{(V,P,I)}(T^{\langle\rangle}_{p_{4.1}}) = (3, \bot)(4, \bot)(7, \{\mathtt{y} = 1\})$, thus it keeps not only the value of variable $\mathtt{y}$ at Line 7 but the path of execution as well. Note that the result of $Proj^{*}_{(V,P,I)}(T^{\langle\rangle}_{q_{4.1}})$ is different because of the different path of execution taken in $q_{4.1}$.

Using the above functions we can define a unified semantic equivalence relation $\mathcal{U}$, which is capable of expressing Korel and Laski's dynamic slicing.

In the following definition, the roles of the parameters are as follows: S denotes the set of initial states for which the equivalence must hold. This captures the 'pure' part of the dynamic slicing critera (the input supplied to the program, or, equivalently the initial state in which the program is to be executed). The set of variables of interest V is common to all slicing criteria. Parameter P, just as in Definitions 4.1 and 4.2, contains the points of interest in the trajectory and it also captures the 'iteration count' component of the criteria. Finally, X captures the 'trajectory requirement'. It is a function that determines which statements must be preserved in the trajectory (even though they have not affected the variables of the slicing criterion). The domain of X is a pair of sets of statement numbers from two programs.

Definition 4.3 (Unified Equivalence)**.** Given programs p and q, a set of states S, a set of variables V, a set of (line number, natural number) pairs P, and a set of line numbers $\times$ set of line numbers $\rightarrow$ set of line numbers function X, the unified equivalence $\mathcal{U}$ is defined as follows:

$$\begin{gathered} p \quad \mathcal{U}(S, V, P, X) \quad q \\ \text{if and only if} \\ \forall \sigma \in S : Proj^*_{(V,P,X(\overline{p},\overline{q}))}(T^\sigma_p) = Proj^*_{(V,P,X(\overline{p},\overline{q}))}(T^\sigma_q) \end{gathered}$$

where $\overline{p}$ and $\overline{q}$ denote the set of statement numbers in p and q, respectively.

4.3 Dynamic and Static Slicing Re-defined

By instantiating Definition 4.3 with appropriate parameters we get a new equivalence relation which captures the semantics of Korel and Laski's dynamic slicing.

Definition 4.4 (Korel and Laski Style Dynamic Equivalence)**.** For a state σ, set of variables V and a (line number, natural number) pair $n^{(k)}$, the Korel-and-Laski-style dynamic equivalence ($\mathcal{D}_{KLi}$) is defined as follows:

$$\mathcal{D}_{KLi}(\sigma, V, n^{(k)}) = \mathcal{U}(\{\sigma\}, V, \{n^{(k)}\}, \cap).$$

We shall adopt the notational convention that a KL subscript indicates that a slicing criterion respects Korel and Laski's requirement (KL1) and an i subscript indicates that only one occurrence of an instruction in the trajectory is of interest. Naturally, $\mathcal{S}$ and $\mathcal{D}$ denote static and dynamic slicing, respectively. Theorem 4.1 establishes that Definition 4.4 faithfully captures Korel and Laski's definition.

Theorem 4.1. *A program p' is a Korel-and-Laski-style dynamic slice of p with respect to the dynamic slicing criterion (x, I^q, V) if and only if p' is a $(\sqsubseteq, \mathcal{D}_{KLi}(\sigma, V, n^{(k)}))$ projection of p, where $\sigma = x$, $n = I$, and q is the position of the kth occurrence of n in T_p^σ.*

First, we will demonstrate the equivalence informally. It is easy to see that the phrase "obtained from p by deleting zero or more statements from it" in Definition 3.10 is equivalent to $\sqsubseteq$. Furthermore, the second and third cases of $Proj'^*$ correspond to the DEL auxiliary function. Finally, the semantics of KL1 (and $Front$) are captured by $Proj^*$; the first case of $Proj'^*$ gives KL2, and from $Proj'^*$ and the definition of l in $Proj^*$ follows KL3.

Now, we will present a formal proof.

Proof. First, we have to show that p' is a $(\sqsubseteq, \mathcal{D}_{KLi}(\sigma, V, n^{(k)}))$ projection of p, assuming that p' is a Korel-and-Laski-style dynamic slice of p with respect to (σ, n^q, V). The fact that a Korel-and-Laski-style dynamic slice is a syntactic subset of the program from which it is constructed implies that $p' \sqsubseteq p$.

Reformulating KL1 gives that a q' exists so that

$$\bigoplus_{i=1}^{q'} \pi(T_{p'}^\sigma(i)) = \bigoplus_{i=1}^{q} \delta(\pi(T_p^\sigma(i))),$$

where $\bigoplus$, as a shorthand notation, denotes the concatenation of sequences, $\pi((m, s)) = (m, \perp)$, and

$$\delta((m, s)) = \begin{cases} (m, s) & \text{if } m \in \overline{p'} \\ \lambda & \text{otherwise.} \end{cases}$$

Since we can apply δ to the left-hand side, from this follows

$$\bigoplus_{i=1}^{q'} \delta(\pi(T^{\sigma}_{p'}(i))) = \bigoplus_{i=1}^{q} \delta(\pi(T^{\sigma}_{p}(i))),$$

where we can substitute $\delta' = \pi \circ \delta$ and thus we get

$$\bigoplus_{i=1}^{q'} \delta'(T^{\sigma}_{p'}(i)) = \bigoplus_{i=1}^{q} \delta'(T^{\sigma}_{p}(i)).$$

From KL2 and KL3 it follows that

$$\delta''(T^{\sigma}_{p'}(q')) = \delta''(T^{\sigma}_{p}(q)),$$

where $\delta''((m,s)) = (m, s \restriction V)$.

If we combine the reformulation of KL1 with that obtained from KL2 and KL3, we get

$$\bigoplus_{i=1}^{q'} \Delta_{q'}(i, T^{\sigma}_{p'}(i)) = \bigoplus_{i=1}^{q} \Delta_{q}(i, T^{\sigma}_{p}(i)),$$

where

$$\Delta_j(i,(m,s)) = \begin{cases} (m, s \restriction V) & \text{if } i = j \\ (m, \bot) & \text{if } i \neq j \text{ and } m \in \overline{p'} \\ \lambda & \text{otherwise.} \end{cases}$$

Since, from KL1 and KL3 it follows that at position q is the kth occurrence of n in T^{σ}_{p} and at position q' is also the kth occurrence of n in $T^{\sigma}_{p'}$, it is clear that the above equation is only a reformulation of $p'\ U(\sigma, V, \{n^{(k)}\}, \cap)\ p$.

Conversely, we also have to show that p' is a Korel-and-Laski-style dynamic slice of p with respect to (σ, n^q, V) if p' is a $(\sqsubseteq, \mathcal{D}_{KLi}(\sigma, V, n^{(k)}))$ projection of p. From $p' \sqsubseteq p$ it follows immediately that p' is a syntactic subset of p. By reformulating $p'\ \mathcal{D}_{KLi}(\sigma, V, n^{(k)})\ p$ we get

$$\bigoplus_{i=1}^{l'} Proj'^{*}_{(V,\{n^{(k)}\},\overline{p'})}(n_i'^{(k_i')}, \sigma_i') = \bigoplus_{i=1}^{l} Proj'^{*}_{(V,\{n^{(k)}\},\overline{p'})}(n_i^{(k_i)}, \sigma_i),$$

where by definition (n_i, σ_i) is $T_p^\sigma(i)$, k_i is the number of occurrences of n_i in the first i elements of T_p^σ and l is the highest index j such that ${n_j}^{(k_j)} \in \{n^{(k)}\}$. Here, n'_i, σ'_i, k'_i and l' have similar meanings in $T_{p'}^\sigma$. From this it follows that l is the position of $n^{(k)}$ in T_p^σ and l' is the position of $n^{(k)}$ in $T_{p'}^\sigma$. Substituting l by q and l' by q', the above equation implies KL1, KL2 and KL3. □

With the help of the unified equivalence not only can we express Korel and Laski's dynamic equivalence, but we can redefine Weiser's static backward equivalence as well.

Definition 4.5 (Traditional Static Equivalence)**.** For a set of variables V and line number n,

$$\mathcal{S}(V, n) = \mathcal{U}(\mathbf{\Sigma}, V, \{n\} \times \mathbf{N}, \varepsilon)$$

where $\mathbf{\Sigma}$ is the set of all possible states, and for every set of line numbers, x and y, $\varepsilon(x, y) = \emptyset$.

Since we have already observed that with an appropriate parameterisation $Proj^*$ reduces to Weiser's $Proj$, it is a trivial matter to show that $\mathcal{S}(V, n)$ is simply a reformulation of the static backward equivalence given in Definition 3.8. The proof here will be left to the reader.

4.4 Eight Forms of Slicing

From the new definitions, we can identify several orthogonal slicing criteria concepts within the slicing criterion. The traditional view of dynamic slicing is that it is obtained from static slicing with the addition of the input sequence to the static slicing criterion. It turns out that this is not the case for KL dynamic slicing. It is more subtle than that. Now, using the projection theory it is possible to tease apart these criterion components.

In the traditional static formulation for slicing, the set of states of interest is the set of all possible states, $\mathbf{\Sigma}$. The set of variables, V and the point in the program n are those of the traditional static slicing criterion. For traditional static slicing, the slicing process must preserve the behaviour of the program

at the point of interest n, and for each possible execution of n (hence $n \times \mathbf{N}$ in Definition 4.5). However, the traditional definition of static slicing places no requirement on the way in which the slice must be computed (hence ε in the same definition). On the other hand, KL–slicing does not require a slice to behave the same way as the original program does for all possible inputs, but only for a specific one, σ. Moreover, the point of interest is only one occurrence of a statement, $n^{(k)}$. Contrary to traditional static slicing, KL–slicing does care about the path of execution in the slice, thus parameter I of $Proj^*$ is $\overline{p} \cap \overline{q}$.

In Figure 4.1, the program performs no input, so the relation $\mathcal{U}$ for this program will not be affected by different choices of the first parameter. So, for any state σ, $p_{4.1}\ \mathcal{U}(\sigma, \{\mathtt{y}\}, \{7^{(1)}\}, \varepsilon)\ q_{4.1}$ but $\neg(p_{4.1}\ \mathcal{U}(\sigma, \{\mathtt{y}\}, \{7^{(1)}\}, \cap)\ q_{4.1})$. That is, the fourth parameter of $\mathcal{U}$, which captures the presence or absence of the KL requirement, is sensitive to the difference in the two programs $p_{4.1}$ and $q_{4.1}$ in Figure 4.1. Observe that for both programs, the final value of $\mathtt{y}$ is 1 regardless of how the program is executed. However, the trajectory followed by the program $q_{4.1}$ differs from the one followed by $p_{4.1}$ even when the two trajectories are restricted to those nodes which occur in both programs; it seems that $q_{4.1}$ arrives at the same answer as $p_{4.1}$ but in a different way.

The requirement that a slice observes this (stringent) requirement for equivalence is similar to the path equivalence studied in the context of program restructuring [65, 88]. It is useful in the context of debugging however. When slicing is applied to debugging, it is vital that the sliced program faithfully reproduce the behaviour that causes a fault to manifest itself as an error. For this reason, program $q_{4.1}$ would not be a useful slice of program $p_{4.1}$ in Figure 4.1. In this regard, the KL requirement is important for debugging applications of slicing [80, 61]. It may also be important in applications to program comprehension [36, 67] because, in these applications, the programmer typically tries to understand the behaviour of the original program in terms of the behaviour of the slice. However, for other applications such as testing, reuse, and restructuring [7, 22, 54], the KL requirement is unimportant because program modification is inherent to these application areas.

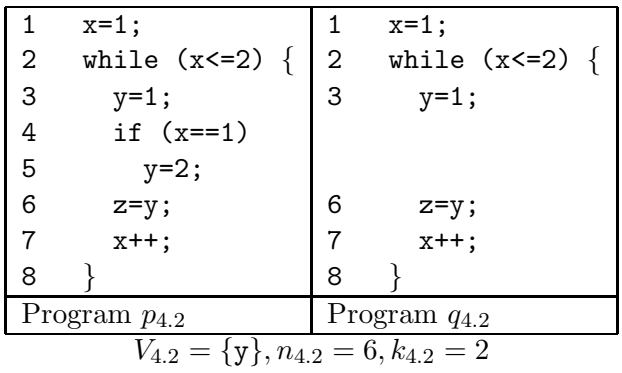


Figure 4.2: Example to capture the difference between iteration count aware and iteration count unaware equivalence relations.

We may also observe that there is another not-yet-discussed concept in the criterion of Korel and Laski's dynamic slicing which is easier to notice when comparing the unified equivalence-based definitions of the traditional static and KL-slicing. Parameter P gives the point(s) of interest in the trajectory in the form of $n^{(k)}$, which has a slightly different meaning from the I^q component of Korel and Laski's original slicing criterion. Theorem 4.1 shows that these two notations are equivalent, and this makes us to realise that the iteration count is a new type of criterion.

To see how the iteration count can affect the meaning of the equivalence preserved by slicing, consider the program in the left-hand column of Figure 4.2. In this program, the conditional at line numbers 4 and 5 can only affect the value of y at Line 6 on the first time it is executed. Therefore, choosing the second iteration of this statement in the slicing criterion will allow the conditional to be deleted. That is, in terms of equivalence, for all states σ, $p_{4.2}\ \mathcal{U}(\sigma, \{\mathtt{y}\}, \{6^{(2)}\}, \cap)\ q_{4.2}$ and $p_{4.2}\ \mathcal{U}(\sigma, \{\mathtt{y}\}, \{6^{(2)}\}, \varepsilon)\ q_{4.2}$.

When slicing is applied to debugging, the iteration count will be of interest. This is because debugging typically starts when the program fails due to a fault. To locate the fault, a slice can be constructed. Of course, it would be sensible to take into account the iteration count for the statement which reveals the error when constructing the slice; this may reduce the size of the slice, thereby reducing the effort involved in debugging.

1 prev=1; 2 curr=1; 3 i=1; 4 while (i<n) { 5 oldc=curr; 6 curr=curr+prev; 7 prev=oldc; 8 i++; 9 }	2 curr=1; 5 oldc=curr; 7 prev=oldc;	1 prev=1; 2 curr=1; 3 i=1; 4 while (i<n) { 5 oldc=curr; 6 curr=curr+prev; 7 prev=oldc; 8 i++; 9 }
Program $p_{4.3}$	$V = \{\texttt{prev}\}, n = 7, k = 1$	$V = \{\texttt{prev}\}, n = 7, k = 2$

Figure 4.3: Example where the iteration count is interesting in a static computation.

Although it was (implicitly) introduced as part of Korel and Laski's dynamic slicing criterion, the iteration count concept is independent of whether a slice is to be static or dynamic. The same is true of the KL requirement. This can be seen from the fact that no input was necessary in the two examples used to illustrate the difference in equivalence relations produced by including or excluding these two requirements. Furthermore, it is possible to find static computations where the iteration count is an interesting and useful concept. For example, in loop carried dependence, it may take several iterations of a loop in order to propagate a dependence from one point to another. An example of this is the program which computes values in the Fibonacci sequence in Figure 4.3. This program performs no input either. In the example, the ability to focus upon different iteration counts allows the dependence structure to be examined in more detail; it becomes possible to see how dependence grows with each loop iteration. In this example, on the first iteration the value of the variable `prev` does not depend on the assignment to `curr` at Line 6, but it depends on the second (and subsequent iterations). As this example demonstrates, the concept of an iteration count may be a useful slicing criterion in its own right.

Finally, consider the example in Figure 4.4, which illustrates the traditional difference between static and dynamic slicing. That is, for dynamic slicing the input affects the outcome of slicing, while for static slicing, the slice

1 y=1; 2 x=input(); 3 if (x>1) 4 y=2; 5 z=y;	1 y=1; 5 z=y;
Program $p_{4.4}$	Program $q_{4.4}$

$\sigma_{4.4} = \langle 1 \rangle, V_{4.4} = \{\mathtt{y}\}, n_{4.4} = 5, k_{4.4} = 1$

Figure 4.4: Example to capture the difference between static and dynamic equivalence relations.

must be correct for all possible initial states. This is the difference between static and dynamic slicing to which most authors [3, 21] refer. However, as the preceding discussion shows, there are two other aspects to a dynamic slice: path equivalence (or otherwise) and iteration count sensitivity (or otherwise).

Now that we have identified the orthogonal criterion components (set of initial states, KL1 restriction or execution path awareness, and iteration count) we realise that the two semantic equivalence relations $\mathcal{S}(V, n)$ and $\mathcal{D}_{KLi}(\sigma, V, n^{(k)})$ represent extremes in a space of eight possible equivalence relations. This space has three orthogonal criteria, which means that there are six additional intervening equivalence relations (and thus, three additional pairs of extremes) resulting from the other possible parameterisations of the unified equivalence. Now, we can define these equivalence relations as well. For the sake of completeness, those relations which have already been presented are included in the list below.

Definition 4.6 (Eight Equivalences)**.**

$$
\begin{array}{lcl}
\mathcal{S}(V,n) & = & \mathcal{U}(\mathbf{\Sigma},V,\{n\}\times\mathbf{N},\varepsilon),\\
\mathcal{S}_i(V,n^{(k)}) & = & \mathcal{U}(\mathbf{\Sigma},V,\{n^{(k)}\},\varepsilon),\\
\mathcal{D}(\sigma,V,n) & = & \mathcal{U}(\{\sigma\},V,\{n\}\times\mathbf{N},\varepsilon),\\
\mathcal{D}_i(\sigma,V,n^{(k)}) & = & \mathcal{U}(\{\sigma\},V,\{n^{(k)}\},\varepsilon),\\
\mathcal{S}_{KL}(V,n) & = & \mathcal{U}(\mathbf{\Sigma},V,\{n\}\times\mathbf{N},\cap),\\
\mathcal{S}_{KLi}(V,n^{(k)}) & = & \mathcal{U}(\mathbf{\Sigma},V,\{n^{(k)}\},\cap),\\
\mathcal{D}_{KL}(\sigma,V,n) & = & \mathcal{U}(\{\sigma\},V,\{n\}\times\mathbf{N},\cap),\\
\mathcal{D}_{KLi}(\sigma,V,n^{(k)}) & = & \mathcal{U}(\{\sigma\},V,\{n^{(k)}\},\cap).
\end{array}
$$

Each of these definitions expresses the semantic aspect of eight different forms of slicing. Six equivalence relations of the above eight capture the semantic property of six new, hitherto undiscussed slicing methods.

Chapter 5

Relationships between Forms of Slicing

5.1 The Subsumes Relation

The eight equivalence relations $\mathcal{S}$, $\mathcal{S}_i$, $\mathcal{D}$, $\mathcal{D}_i$, $\mathcal{S}_{KL}$, $\mathcal{S}_{KLi}$, $\mathcal{D}_{KL}$ and $\mathcal{D}_{KLi}$ in fact represent classes of equivalence relations, since they are parameterised by σ, V, n and k (even though not all of the relations make use of all four). Denoting a parameterised equivalence relation by $\approx$, it is possible to define a subsumption relationship $\approx_B \subseteq \approx_A$ between these classes.

Definition 5.1 (Subsumes Relation)**.** For equivalence relations $\approx_A$ and $\approx_B$, both parameterised by σ, V, n and k, $\approx_A$ subsumes $\approx_B$, denoted as $\approx_B \subseteq \approx_A$, if and only if

$$\forall \sigma, V, n, k : \approx_B^{(\sigma,V,n,k)} \subseteq \approx_A^{(\sigma,V,n,k)}$$

or equivalently,

$$\forall p, q, \sigma, V, n, k : (p, q) \in \approx_B^{(\sigma,V,n,k)} \Rightarrow (p, q) \in \approx_A^{(\sigma,V,n,k)} .$$

This subsumes relation is a partial ordering of parameterised equivalence relations, since it is defined with the help of the subset relation, which is itself a partial ordering (i.e., reflexive, transitive and antisymmetric). Figure 5.1 presents the lattice of the subsumes relation for $\mathcal{S}$, $\mathcal{S}_i$, $\mathcal{D}$, $\mathcal{D}_i$, $\mathcal{S}_{KL}$,

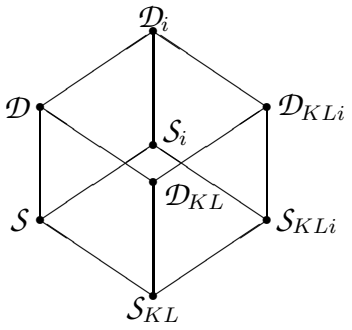


Figure 5.1: Subsumes relationship between equivalence relations.

$\mathcal{S}_{KLi}$, $\mathcal{D}_{KL}$ and $\mathcal{D}_{KLi}$ (e.g., $\mathcal{S}$ is subsumed by $\mathcal{D}$). As can be seen, the relationship between the semantic aspect of static and dynamic slicing is not as straightforward as previous authors have claimed [21, 41, 100]. The following theorem proves the correctness of the diagram in Figure 5.1: in other words, there are no superfluous edges and no missing edges in the lattice.

Theorem 5.1. *The lattice shown in Figure 5.1 is correct: two parameterised equivalence relations are connected in the diagram if and only if they are in subsumes relation.*

The proof makes use of four lemmas and their corollaries. The first lemma is used to prove the "if" direction and the latter three the "only if" direction.

Lemma 5.2. *Given sets of initial states S_1 and S_2, sets of variables V_1 and V_2, sets of points of interests P_1 and P_2 and functions of pairs of line number sets X_1 and X_2 such that*

$$S_1 \subseteq S_2,\ \ V_1 \subseteq V_2,\ \ P_1 \subseteq P_2,\quad \textit{and}\ \ \forall p, q : X_1(\overline{p}, \overline{q})\ \subseteq\ X_2(\overline{p}, \overline{q})$$
$$\textit{then}$$
$$\mathcal{U}(S_2, V_2, P_2, X_2)\ \subseteq\ \mathcal{U}(S_1, V_1, P_1, X_1).$$

Proof. Let $(p, q) \in \mathcal{U}(S_2, V_2, P_2, X_2)$. By definition, this is equivalent to $\forall \sigma \in S_2$:

$$Proj^*_{(V_2, P_2, X_2(\overline{p}, \overline{q}))}(T^\sigma_p) = Proj^*_{(V_2, P_2, X_2(\overline{p}, \overline{q}))}(T^\sigma_q).$$

As $S_1 \subseteq S_2$, it follows that $\forall \sigma \in S_1$:

$$Proj^*_{(V_2,P_2,X_2(\overline{p},\overline{q}))}(T^\sigma_p) = Proj^*_{(V_2,P_2,X_2(\overline{p},\overline{q}))}(T^\sigma_q).$$

By inlining the definition of $Proj^*$, this is equivalent to $\forall \sigma \in S_1$:

$$\bigoplus_{i=1}^{l_2^p} Proj'^*_{(V_2,P_2,X_2(\overline{p},\overline{q}))}(n_i^{p(k_i^p)}, \sigma_i^p) = \bigoplus_{i=1}^{l_2^q} Proj'^*_{(V_2,P_2,X_2(\overline{p},\overline{q}))}(n_i^{q(k_i^q)}, \sigma_i^q).$$

Corresponding prefixes from the above equality are also equal. As the points of interest P_1 is a subset of P_2, the location of the last occurrence in the trajectory of a point from P_1 must occur before the last occurrence of a point from P_2. More formally, $l_1^p \leq l_2^p$ and $l_1^q \leq l_2^q$; thus, the above equality holds when $\bigoplus_{i=1}^{l_2^p}$ and $\bigoplus_{i=1}^{l_2^q}$ are replaced by $\bigoplus_{i=1}^{l_1^p}$ and $\bigoplus_{i=1}^{l_1^q}$. Consequently, $\forall \sigma \in S_1$:

$$\bigoplus_{i=1}^{l_1^p} Proj'^*_{(V_2,P_2,X_2(\overline{p},\overline{q}))}(n_i^{p(k_i^p)}, \sigma_i^p) = \bigoplus_{i=1}^{l_1^q} Proj'^*_{(V_2,P_2,X_2(\overline{p},\overline{q}))}(n_i^{q(k_i^q)}, \sigma_i^q).$$

This means that the projections of those state trajectory elements, which are not projected to λ are pairwise equal in the two trajectories. Thus, any corresponding subsequence of these elements must be pairwise equal. In particular, as $V_1 \subseteq V_2$, $P_1 \subseteq P_2$, and $\forall p, q : X_1(\overline{p},\overline{q}) \subseteq X_2(\overline{p},\overline{q})$, restricting the sequences to variables in V_1 at points in P_1 where the instruction from $X_1(\overline{p},\overline{q})$ are preserved must also be equivalent. Thus, $\forall \sigma \in S_1$:

$$\bigoplus_{i=1}^{l_1^p} Proj'^*_{(V_1,P_1,X_1(\overline{p},\overline{q}))}(n_i^{p(k_i^p)}, \sigma_i^p) = \bigoplus_{i=1}^{l_1^q} Proj'^*_{(V_1,P_1,X_1(\overline{p},\overline{q}))}(n_i^{q(k_i^q)}, \sigma_i^q),$$

which, by definition, is equivalent to $\forall \sigma \in S_1$:

$$Proj^*_{(V_1,P_1,X_1(\overline{p},\overline{q}))}(T^\sigma_p) = Proj^*_{(V_1,P_1,X_1(\overline{p},\overline{q}))}(T^\sigma_q).$$

which, again by definition, is equivalent to $(p, q) \in \mathcal{U}(S_1, V_1, P_1, X_1)$, as required. □

The existence of each of the 12 subsumption relationships between the parameterised equivalence relations shown in Figure 5.1 follows from Lemma 5.2, as proven by the corollary below.

Corollary 5.3. *The parameterised equivalence relations connected in the diagram are in subsumes relation.*

Proof. The proof of each case considers each of the four attributes of the unified equivalence operator $\mathcal{U}(S, V, P, X)$ independently. The relevant relationships are as follows: For S, $\{\sigma\} \subseteq \mathbf{\Sigma}$, for V, all 12 use the same argument (which is thus ignored below), for P, $n^{(k)} \subseteq (\{n\} \times \mathbf{N})$, and for X, $\forall p, q : \varepsilon(\overline{p}, \overline{q}) = \emptyset \subseteq \cap(\overline{p}, \overline{q})$. The table below shows how Lemma 5.2 implies all of the cases ($\varepsilon(\overline{p}, \overline{q})$ and $\cap(\overline{p}, \overline{q})$ have been abbreviated ε and $\cap$).

Subsumption	Lemma 5.2 Requirement		
	S	P	X
$\mathcal{D}_{KLi} \subseteq \mathcal{D}_i$	$\{\sigma\} \subseteq \{\sigma\}$	$n^{(k)} \subseteq n^{(k)}$	$\varepsilon \subseteq \cap$
$\mathcal{D}_{KL} \subseteq \mathcal{D}_{KLi}$	$\{\sigma\} \subseteq \{\sigma\}$	$n^{(k)} \subseteq \{n\} \times \mathbf{N}$	$\cap \subseteq \cap$
$\mathcal{D}_{KL} \subseteq \mathcal{D}$	$\{\sigma\} \subseteq \{\sigma\}$	$\{n\} \times \mathbf{N} \subseteq \{n\} \times \mathbf{N}$	$\varepsilon \subseteq \cap$
$\mathcal{D} \subseteq \mathcal{D}_i$	$\{\sigma\} \subseteq \{\sigma\}$	$n^{(k)} \subseteq \{n\} \times \mathbf{N}$	$\varepsilon \subseteq \varepsilon$
$\mathcal{S}_{KLi} \subseteq \mathcal{D}_{KLi}$	$\{\sigma\} \subseteq \mathbf{\Sigma}$	$n^{(k)} \subseteq n^{(k)}$	$\cap \subseteq \cap$
$\mathcal{S}_{KLi} \subseteq \mathcal{S}_i$	$\mathbf{\Sigma} \subseteq \mathbf{\Sigma}$	$n^{(k)} \subseteq n^{(k)}$	$\varepsilon \subseteq \cap$
$\mathcal{S}_{KL} \subseteq \mathcal{D}_{KL}$	$\{\sigma\} \subseteq \mathbf{\Sigma}$	$\{n\} \times \mathbf{N} \subseteq \{n\} \times \mathbf{N}$	$\cap \subseteq \cap$
$\mathcal{S}_{KL} \subseteq \mathcal{S}_{KLi}$	$\mathbf{\Sigma} \subseteq \mathbf{\Sigma}$	$n^{(k)} \subseteq \{n\} \times \mathbf{N}$	$\cap \subseteq \cap$
$\mathcal{S}_{KL} \subseteq \mathcal{S}$	$\mathbf{\Sigma} \subseteq \mathbf{\Sigma}$	$\{n\} \times \mathbf{N} \subseteq \{n\} \times \mathbf{N}$	$\varepsilon \subseteq \cap$
$\mathcal{S}_i \subseteq \mathcal{D}_i$	$\{\sigma\} \subseteq \mathbf{\Sigma}$	$n^{(k)} \subseteq n^{(k)}$	$\varepsilon \subseteq \varepsilon$
$\mathcal{S} \subseteq \mathcal{D}$	$\{\sigma\} \subseteq \mathbf{\Sigma}$	$\{n\} \times \mathbf{N} \subseteq \{n\} \times \mathbf{N}$	$\varepsilon \subseteq \varepsilon$
$\mathcal{S} \subseteq \mathcal{S}_i$	$\mathbf{\Sigma} \subseteq \mathbf{\Sigma}$	$n^{(k)} \subseteq \{n\} \times \mathbf{N}$	$\varepsilon \subseteq \varepsilon$

□

The proof of the "only if" direction involves showing that there are no "missing" edges in Figure 5.1. To be more precise, the following nine pairs of parameterised slicing equivalence relations are incomparable (denoted by "$\approx_A \not\supseteq\not\subseteq \approx_B$"): ($\mathcal{D} \not\supseteq\not\subseteq \mathcal{D}_{KLi}$), ($\mathcal{D} \not\supseteq\not\subseteq \mathcal{S}_i$), ($\mathcal{D}_{KLi} \not\supseteq\not\subseteq \mathcal{S}_i$), ($\mathcal{S}_i \not\supseteq\not\subseteq \mathcal{D}_{KL}$),

($\mathcal{D}_{KL} \not\supseteq\not\subseteq \mathcal{S}$), ($\mathcal{D}_{KL} \not\supseteq\not\subseteq \mathcal{S}_{KLi}$), ($\mathcal{S} \not\supseteq\not\subseteq \mathcal{S}_{KLi}$), ($\mathcal{D} \not\supseteq\not\subseteq \mathcal{S}_{KLi}$), and ($\mathcal{D}_{KLi} \not\supseteq\not\subseteq \mathcal{S}$).

Proving the incomparability of two parameterised equivalence relations $\approx_A$ and $\approx_B$ requires showing that neither relation subsumes the other. This is done by showing two things. First, that there exist programs p and q such that $(p,q) \in \approx_A^{(\sigma,V,n,k)}$ but $(p,q) \notin \approx_B^{(\sigma,V,n,k)}$ (for some σ, V, n and k that parameterise the relations) and then by showing that there exist programs p' and q' such that $(p',q') \in \approx_B^{(\sigma',V',n',k')}$ but $(p',q') \notin \approx_A^{(\sigma',V',n',k')}$ (for some other σ', V', n' and k'). The following three lemmas introduce examples used to show the necessary incomparabilities.

Lemma 5.4. $\mathcal{S} \not\subseteq \mathcal{D}_{KLi}$

Proof. For $p_{4.1}$ and $q_{4.1}$ as given in Figure 4.1, for $V_{4.1} = \{\mathtt{y}\}, n_{4.1} = 7, k_{4.1} = 1$ and for any input state σ the following is shown:

$$(p_{4.1}, q_{4.1}) \in \mathcal{S}(V_{4.1}, n_{4.1}) \text{ and } (p_{4.1}, q_{4.1}) \notin \mathcal{D}_{KLi}(\sigma, V_{4.1}, {n_{4.1}}^{(k_{4.1})}).$$

First, as the program is unaffected by its input, $(p_{4.1}, q_{4.1}) \in \mathcal{S}(V_{4.1}, n_{4.1})$ for all input states σ because $\forall \sigma \in \mathbf{\Sigma}$:

$$Proj^*_{(V_{4.1},\{n_{4.1}\}\times\mathbf{N},\emptyset)}(T^{\sigma}_{p_{4.1}}) = Proj^*_{(V_{4.1},\{n_{4.1}\}\times\mathbf{N},\emptyset)}(T^{\sigma}_{q_{4.1}}) = (7, \{\mathtt{y} = 1\}).$$

Second $(p_{4.1}, q_{4.1}) \notin \mathcal{D}_{KLi}(\sigma, V_{4.1}, {n_{4.1}}^{(k_{4.1})})$ because, as shown in Section 4.2, KL1 is violated. Thus, combined $\mathcal{S}(V_{4.1}, n_{4.1}) \not\subseteq \mathcal{D}_{KLi}(\sigma, V_{4.1}, {n_{4.1}}^{(k_{4.1})})$ and, in general, $\mathcal{S} \not\subseteq \mathcal{D}_{KLi}$. □

Five corollaries to Lemma 5.4 are used in the proof of Theorem 5.1. They are given as Equations (2) through (6) in Figure 5.2. A detailed proof of the first is given below; the other proofs are similar. Note that Equation (6) follows from Equations (2) and (4).

Corollary 5.5. $\mathcal{S}_i \not\subseteq \mathcal{D}_{KLi}$ *(Equation (2) of Figure 5.2)*

Proof. From Lemma 5.4 we know that $(p_{4.1}, q_{4.1}) \in \mathcal{S}(V_{4.1}, n_{4.1})$ and $(p_{4.1}, q_{4.1}) \notin \mathcal{D}_{KLi}(\sigma, V_{4.1}, {n_{4.1}}^{(k_{4.1})})$, for any $\sigma \in \mathbf{\Sigma}$. Additionally, Lemma 5.2 implies

$\mathcal{S}(V_{4.1}, n_{4.1}) \subseteq \mathcal{S}_i(V_{4.1}, {n_{4.1}}^{(k_{4.1})})$, from which follows $(p_{4.1}, q_{4.1}) \in \mathcal{S}_i(V_{4.1}, {n_{4.1}}^{(k_{4.1})})$. Thus, it must be the case that $\mathcal{S}_i(V_{4.1}, n_{4.1}) \not\subseteq \mathcal{D}_{KLi}(\sigma, V_{4.1}, {n_{4.1}}^{(k_{4.1})})$ or simply $\mathcal{S}_i \not\subseteq \mathcal{D}_{KLi}$. □

Lemma 5.6. $\mathcal{S}_{KLi} \not\subseteq \mathcal{D}$

Proof. For $p_{4.2}, q_{4.2}, V_{4.2}, n_{4.2}, k_{4.2}$ as given in Figure 4.2, and for any input state σ the following hold:

$$(p_{4.2}, q_{4.2}) \in \mathcal{S}_{KLi}(V_{4.2}, {n_{4.2}}^{(k_{4.2})}) \text{ and } (p_{4.2}, q_{4.2}) \notin \mathcal{D}(\sigma, V_{4.2}, n_{4.2}).$$

First, $(p_{4.2}, q_{4.2}) \in \mathcal{S}_{KLi}(V_{4.2}, {n_{4.2}}^{(k_{4.2})})$ since $\forall \sigma \in \mathbf{\Sigma}$:

$$\begin{aligned} &Proj^*_{(V_{4.2}, \{{n_{4.2}}^{(k_{4.2})}\}, \overline{p_{4.2}} \cap \overline{q_{4.2}})}(T^\sigma_{p_{4.2}}) \\ &\quad = Proj^*_{(V_{4.2}, \{{n_{4.2}}^{(k_{4.2})}\}, \overline{p_{4.2}} \cap \overline{q_{4.2}})}(T^\sigma_{q_{4.2}}) \\ &\quad = (2, \perp)(3, \perp)(6, \perp)(7, \perp)(2, \perp)(3, \perp)(6, \{\mathtt{y} = 1\}). \end{aligned}$$

Second $(p_{4.2}, q_{4.2}) \notin \mathcal{D}(\sigma, V_{4.2}, n_{4.2})$ because

$$Proj^*_{(V_{4.2}, \{n_{4.2}\} \times \mathbf{N}, \emptyset)}(T^\sigma_{p_{4.2}}) = (6, \{\mathtt{y} = 2\})(6, \{\mathtt{y} = 1\}),$$

but

$$Proj^*_{(V_{4.2}, \{n_{4.2}\} \times \mathbf{N}, \emptyset)}(T^\sigma_{q_{4.2}}) = (6, \{\mathtt{y} = 1\})(6, \{\mathtt{y} = 1\}).$$

Thus, combined $\mathcal{S}_{KLi}(V_{4.2}, {n_{4.2}}^{(k_{4.2})}) \not\subseteq \mathcal{D}(\sigma, V_{4.2}, n_{4.2})$ and, in general, $\mathcal{S}_{KLi} \not\subseteq \mathcal{D}$. □

As with Lemma 5.4, five corollaries to Lemma 5.6 are given as Equations (8) through (12) in Figure 5.2. Note that Equation (12) follows from Equations (9) and (11).

Lemma 5.7. $\mathcal{D}_{KL} \not\subseteq \mathcal{S}_i$

Proof. For $p_{4.4}, q_{4.4}, \sigma_{4.4}, V_{4.4}, n_{4.4}, k_{4.4}$ as given in Figure 4.4, the following must be shown to be true:

$$(p_{4.4}, q_{4.4}) \in \mathcal{D}_{KL}(\sigma_{4.4}, V_{4.4}, n_{4.4}) \text{ and } (p_{4.4}, q_{4.4}) \notin \mathcal{S}_i(V_{4.4}, {n_{4.4}}^{(k_{4.4})}).$$

Eq.	Result/Corollaries	Justification
(1)	$\mathcal{S} \not\subseteq \mathcal{D}_{KLi}$ by Lemma 5.4	
(2)	$\mathcal{S}_i \not\subseteq \mathcal{D}_{KLi}$ as $\mathcal{S} \subseteq \mathcal{S}_i$	implies $(p_{4.1}, q_{4.1}) \in \mathcal{S}_i(V_{4.1}, {n_{4.1}}^{(k_{4.1})})$
(3)	$\mathcal{D} \not\subseteq \mathcal{D}_{KLi}$ as $\mathcal{S} \subseteq \mathcal{D}$	implies $(p_{4.1}, q_{4.1}) \in \mathcal{D}(\sigma, V_{4.1}, n_{4.1})$
(4)	$\mathcal{S} \not\subseteq \mathcal{D}_{KL}$ as $\mathcal{D}_{KL} \subseteq \mathcal{D}_{KLi}$	implies $(p_{4.1}, q_{4.1}) \notin \mathcal{D}_{KL}(\sigma, V_{4.1}, n_{4.1})$
(5)	$\mathcal{S} \not\subseteq \mathcal{S}_{KLi}$ as $\mathcal{S}_{KLi} \subseteq \mathcal{D}_{KLi}$	implies $(p_{4.1}, q_{4.1}) \notin \mathcal{S}_{KLi}(V_{4.1}, {n_{4.1}}^{(k_{4.1})})$
(6)	$\mathcal{S}_i \not\subseteq \mathcal{D}_{KL}$ as $(p_{4.1}, q_{4.1}) \in \mathcal{S}_i(V_{4.1}, {n_{4.1}}^{(k_{4.1})})$ and $(p_{4.1}, q_{4.1}) \notin \mathcal{D}_{KL}(\sigma, V_{4.1}, n_{4.1})$	
(7)	$\mathcal{S}_{KLi} \not\subseteq \mathcal{D}$ by Lemma 5.6	
(8)	$\mathcal{S}_i \not\subseteq \mathcal{D}$ as $\mathcal{S}_{KLi} \subseteq \mathcal{S}_i$	implies $(p_{4.2}, q_{4.2}) \in \mathcal{S}_i(V_{4.2}, {n_{4.2}}^{(k_{4.2})})$
(9)	$\mathcal{D}_{KLi} \not\subseteq \mathcal{D}$ as $\mathcal{S}_{KLi} \subseteq \mathcal{D}_{KLi}$	implies $(p_{4.2}, q_{4.2}) \in \mathcal{D}_{KLi}(\sigma, V_{4.2}, {n_{4.2}}^{(k_{4.2})})$
(10)	$\mathcal{S}_{KLi} \not\subseteq \mathcal{D}_{KL}$ as $\mathcal{D}_{KL} \subseteq \mathcal{D}$	implies $(p_{4.2}, q_{4.2}) \notin \mathcal{D}_{KL}(\sigma, V_{4.2}, n_{4.2})$
(11)	$\mathcal{S}_{KLi} \not\subseteq \mathcal{S}$ as $\mathcal{S} \subseteq \mathcal{D}$	implies $(p_{4.2}, q_{4.2}) \notin \mathcal{S}(V_{4.2}, n_{4.2})$
(12)	$\mathcal{D}_{KLi} \not\subseteq \mathcal{S}$ as $(p_{4.2}, q_{4.2}) \in \mathcal{D}_{KLi}(\sigma, V_{4.2}, {n_{4.2}}^{(k_{4.2})})$ and $(p_{4.2}, q_{4.2}) \notin \mathcal{S}(V_{4.2}, n_{4.2})$	
(13)	$\mathcal{D}_{KL} \not\subseteq \mathcal{S}_i$ by Lemma 5.7	
(14)	$\mathcal{D} \not\subseteq \mathcal{S}_i$ as $\mathcal{D}_{KL} \subseteq \mathcal{D}$	implies $(p_{4.4}, q_{4.4}) \in \mathcal{D}(\sigma_{4.4}, V_{4.4}, n_{4.4})$
(15)	$\mathcal{D}_{KLi} \not\subseteq \mathcal{S}_i$ as $\mathcal{D}_{KL} \subseteq \mathcal{D}_{KLi}$	implies $(p_{4.4}, q_{4.4}) \in \mathcal{D}_{KLi}(\sigma_{4.4}, V_{4.4}, {n_{4.4}}^{(k_{4.4})})$
(16)	$\mathcal{D}_{KL} \not\subseteq \mathcal{S}$ as $\mathcal{S} \subseteq \mathcal{S}_i$	implies $(p_{4.4}, q_{4.4}) \notin \mathcal{S}(V_{4.4}, n_{4.4})$
(17)	$\mathcal{D}_{KL} \not\subseteq \mathcal{S}_{KLi}$ as $\mathcal{S}_{KLi} \subseteq \mathcal{S}_i$	implies $(p_{4.4}, q_{4.4}) \notin \mathcal{S}_{KLi}(V_{4.4}, {n_{4.4}}^{(k_{4.4})})$
(18)	$\mathcal{D} \not\subseteq \mathcal{S}_{KLi}$ as $(p_{4.4}, q_{4.4}) \in \mathcal{D}(\sigma_{4.4}, V_{4.4}, n_{4.4})$ and $(p_{4.4}, q_{4.4}) \notin \mathcal{S}_{KLi}(V_{4.4}, {n_{4.4}}^{(k_{4.4})})$	

Figure 5.2: Corollaries to Lemmas 5.4, 5.6, and 5.7.

First, $(p_{4.4}, q_{4.4}) \in \mathcal{D}_{KL}(\sigma_{4.4}, V_{4.4}, n_{4.4})$ since

$$\begin{aligned} &Proj^*_{(V_{4.4}, \{n_{4.4}\} \times \mathbf{N}, \overline{p_{4.4}} \cap \overline{q_{4.4}})}(T^{\sigma_{4.4}}_{p_{4.4}}) \\ &\quad = Proj^*_{(V_{4.4}, \{n_{4.4}\} \times \mathbf{N}, \overline{p_{4.4}} \cap \overline{q_{4.4}})}(T^{\sigma_{4.4}}_{q_{4.4}}) \\ &\quad = (5, \{\mathtt{y} = 1\}). \end{aligned}$$

Second $(p_{4.4}, q_{4.4}) \notin \mathcal{S}_i(V_{4.4}, {n_{4.4}}^{(k_{4.4})})$ because $Proj^*_{(V_{4.4}, \{{n_{4.4}}^{(k_{4.4})}\}, \emptyset)}(T^{\sigma^*}_{p_{4.4}}) = (5, \{\mathtt{y} = 2\})$ but $Proj^*_{(V_{4.4}, \{{n_{4.4}}^{(k_{4.4})}\}, \emptyset)}(T^{\sigma^*}_{q_{4.4}}) = (5, \{\mathtt{y} = 1\})$, where $\sigma^* = \langle 2 \rangle$. Thus, combined $\mathcal{S}_i(V_{4.4}, {n_{4.4}}^{(k_{4.4})}) \not\subseteq \mathcal{D}_{KL}(\sigma_{4.4}, V_{4.4}, n_{4.4})$, and in general, $\mathcal{S}_i \not\subseteq \mathcal{D}_{KL}$. □

As with Lemmas 5.4 and 5.6, five corollaries to Lemma 5.7 are given as Equations (14) through (18) in Figure 5.2. Note that Equation (18) follows from Equations (14) and (17). Using Lemma 5.2 and Equations (1) through (18) from Figure 5.2, it is now possible to prove Theorem 5.1, which is restated.

Theorem 5.1. *The lattice shown in Figure 5.1 is correct: two parameterised equivalence relations are connected in the diagram if and only if they are in subsumes relation.*

Proof. The "if" direction is proven in Corollary 5.3, while the relations given in Figure 5.2 are sufficient to prove all of the cases in the "only if" direction, as summarised in the following table:

Incompatibility	Follows from Figure 5.2 Equations
$\mathcal{D} \not\supseteq\not\subseteq \mathcal{D}_{KLi}$	(3) and (9)
$\mathcal{D} \not\supseteq\not\subseteq \mathcal{S}_i$	(14) and (8)
$\mathcal{D}_{KLi} \not\supseteq\not\subseteq \mathcal{S}_i$	(15) and (2)
$\mathcal{S}_i \not\supseteq\not\subseteq \mathcal{D}_{KL}$	(6) and (13)
$\mathcal{D}_{KL} \not\supseteq\not\subseteq \mathcal{S}$	(16) and (4)
$\mathcal{D}_{KL} \not\supseteq\not\subseteq \mathcal{S}_{KLi}$	(17) and (10)
$\mathcal{S} \not\supseteq\not\subseteq \mathcal{S}_{KLi}$	(5) and (11)
$\mathcal{D} \not\supseteq\not\subseteq \mathcal{S}_{KLi}$	(18) and (7)
$\mathcal{D}_{KLi} \not\supseteq\not\subseteq \mathcal{S}$	(12) and (1)

□

5.2 Subsumes Relation of Slicing Techniques

In the above we studied the relationships between the *semantic* properties of eight forms of slicing. In general, however, in addition to studying the semantic properties of slicing, we are also interested in the relationship between the forms of slicing, not merely in the relationships between the semantic equivalence relations.

In order to achieve this, we will need to take account of both the syntactic ordering relation and the semantic equivalence relation. We call the combination of a syntactic ordering and a parameterised equivalence a slicing technique, and we define subsumes relations between the slicing techniques as well (e.g., between static and Korel and Laski's dynamic slicing).

Informally, a slicing technique s_1 subsumes another slicing technique s_2 if and only if all slices of an arbitrary program with respect to any given slicing criterion according to s_2 are valid slices with respect to the same slicing criterion according to s_1. This informal definition is formalised below.

Definition 5.2 (Subsumes Relation of Slicing Techniques)**.** Given syntactic ordering $\lesssim$ and semantic equivalence relations $\approx_A$ and $\approx_B$, both parameterised by σ, V, n and k, $(\lesssim, \approx_A)$–slicing subsumes $(\lesssim, \approx_B)$–slicing if and only if

$$\forall p, \sigma, V, n, k : \mathbb{S}_p(\lesssim, \approx_B^{(\sigma,V,n,k)}) \subseteq \mathbb{S}_p(\lesssim, \approx_A^{(\sigma,V,n,k)}),$$

where $\mathbb{S}_p(\lesssim, \approx) = \{q | q \approx p \text{ and } q \lesssim p\}$ is the set of all the possible slices of program p for given projection $(\lesssim, \approx)$.

For example, $(\sqsubseteq, \mathcal{S}_i)$–slicing subsumes $(\sqsubseteq, \mathcal{S})$–slicing as every $(\sqsubseteq, \mathcal{S}(V, n))$ projection of a given program p is a $(\sqsubseteq, \mathcal{S}_i(V, n^{(k)}))$ projection of p as well, i.e. $\mathbb{S}_p(\sqsubseteq, \mathcal{S}(V, n)) \subseteq \mathbb{S}_p(\sqsubseteq, \mathcal{S}_i(V, n^{(k)}))$, for any given V, n and k. On the contrary, $(\sqsubseteq, \mathcal{S})$–slicing does not subsume $(\sqsubseteq, \mathcal{S}_i)$–slicing. This is illustrated in Figure 4.2 where $q_{4.2}$ is a $(\sqsubseteq, \mathcal{S}_i(\{\mathtt{y}\}, 6^{(2)}))$ projection of $p_{4.2}$ but it is not a $(\sqsubseteq, \mathcal{S}(\{\mathtt{y}\}, 6))$ projection.

This definition of subsumption relationship between slicing techniques is closely related to the subsumption relationship defined for parameterised semantic equivalence relations. Namely, if $\approx_A$ subsumes $\approx_B$ then $(\lesssim, \approx_A)$–slicing subsumes $(\lesssim, \approx_B)$–slicing as well. This is stated and proven in the following lemma:

Lemma 5.8. *Given semantic equivalence relations $\approx_A$ and $\approx_B$, both parameterised with σ, V, n and k, if $\approx_A$ subsumes $\approx_B$ then $(\lesssim, \approx_A)$–slicing subsumes $(\lesssim, \approx_B)$–slicing, for any syntactic ordering $\lesssim$.*

Proof. Let p be a program, $\lesssim$ a syntactic ordering and $q \in \mathbb{S}_p(\lesssim, \approx_B^{(\sigma,V,n,k)})$ (for any given σ, V, n and k). Then, by definition, $q \approx_B^{(\sigma,V,n,k)} p$ and $q \lesssim p$. Since $\approx_A$ subsumes $\approx_B$, $q \approx_A^{(\sigma,V,n,k)} p$ holds as well, which means that $q \in \mathbb{S}_p(\lesssim, \approx_A^{(\sigma,V,n,k)})$ as required. □

The above lemma can be used to prove the correctness of the diagram depicted in Figure 5.3, which mirrors the diagram from Figure 5.1. Figure 5.3

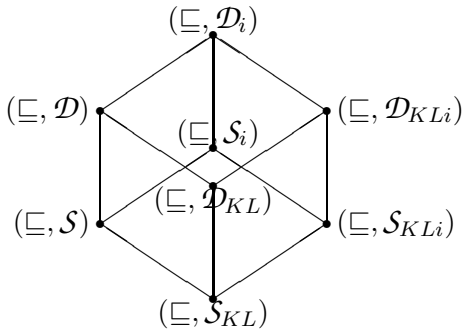


Figure 5.3: Subsumes relationship between slicing techniques.

shows the precise connections between the slicing techniques (as opposed to equivalence relations) that all use the traditional syntactic ordering $\sqsubseteq$ and the parameterised equivalence relations $\mathcal{S}$, $\mathcal{S}_i$, $\mathcal{D}$, $\mathcal{D}_i$, $\mathcal{S}_{KL}$, $\mathcal{S}_{KLi}$, $\mathcal{D}_{KL}$ and $\mathcal{D}_{KLi}$. The correctness of this diagram is shown in the following theorem.

Theorem 5.9. *The lattice shown in Figure 5.3 is correct: two slicing techniques are connected in the diagram if and only if they are in subsumes relation.*

Proof. "if": The correctness of each of the subsumption relations shown in Figure 5.3 follows from Theorem 5.1 and Lemma 5.8 where the syntactic ordering is $\sqsubseteq$.

"only if": The argument that no edges are missing from the diagram follows from the "only if" argument of Theorem 5.1 and the observation that the examples in Figures 4.1, 4.2, and 4.4 are constructed so that $q_{4.1} \sqsubseteq p_{4.1}$, $q_{4.2} \sqsubseteq p_{4.2}$ and $q_{4.4} \sqsubseteq p_{4.4}$. □

Theorems 5.1 and 5.9 formalise the relationship between the eight equivalence relations and the derived slicing techniques depicted in Figures 5.1 and 5.3, respectively. The significance of this result is that it shows that the dynamic slicing criterion contains two, previously un-studied criteria: path sensitivity and iteration count sensitivity. The presence of these criteria make the subsumption relationship between the forms of static and dynamic slicing more involved than previously thought. This is both theoretically interesting

and practically important. As mentioned in the previous chapter, the new criteria may also find useful applications in their own right. For example, they allow those working on building slicers to better understand the trade-offs between slicing precision and computation time. They also allow slice users to understand and then choose the most appropriate slicing definition for a given problem.

5.3 Minimal Slices

Statements like "dynamic slices are smaller than static slices" are occasionally heard amongst slicing researchers. We intuitively know what is meant by such statements but clearly, not every dynamic slice *is* smaller than every static slice. Even for a given choice of program point and variable, the statement may not be true, because of differences in slicing algorithms. Furthermore, there is the complication of which particular dynamic slicing definition one is to adopt; some are incomparable with static slicing. One interpretation of what is meant by such statements is that the *minimal* slices inherent in dynamic slicing are smaller than the *minimal* slices inherent in static slicing.

To compare slicing techniques, it is important to be free of the algorithmic and implementation details. We are concerned with the investigation of various definitions for 'slice'; not the peculiarities which emerge from attempts to arrive at 'good' slicing algorithms. In other words, we are concerned with the output of idealised algorithms. Any realisable slicing algorithm must by definition compute an *approximation* to the idealised algorithm. Even with idealised algorithms there is no guarantee of a unique minimal slice. Therefore, sets of minimal slices will be studied. Such a set includes all the 'best' (i.e., smallest) slices.

To formalise the beliefs about the size of slices (more precisely, about the size of minimal slices), we shall compare sets of minimal slices. To allow such a comparison, we have to extend the syntactic ordering of programs (from Definition 3.1) to sets of programs.

Definition 5.3 (Extending $\lesssim$ to Sets)**.** Given a pre-order $\lesssim$ over a set, we

can define a pre-order over its subsets as follows

$$A \lesssim B \iff \forall b \in B : \exists a \in A : a \lesssim b.$$

Now we will show that the syntactic ordering extended to sets from Definition 5.3 is indeed a pre-order.

Lemma 5.10. *The syntactic ordering is a pre-order over sets of programs.*

Proof. We have to show that the relation given in Definition 5.3 is reflexive and transitive.

Reflexivity. We have to show that $A \lesssim A$ holds for all sets of programs. According to the definition this means that we have to show that $\forall a \in A : \exists a' \in A : a' \lesssim a$. This follows as $a \lesssim a$.

Transitivity. We have to show for all A, B and C sets of programs that $A \lesssim B$ and $B \lesssim C$ imply $A \lesssim C$. That is, we have to show that $\forall c \in C : \exists a \in A : a \lesssim c$. From $B \lesssim C$ we know that $\forall c \in C : \exists b \in B : b \lesssim c$. Furthermore, from $A \lesssim B$ we know that $\forall b \in B : \exists a \in A : a \lesssim b$. Together these imply $\forall c \in C : \exists b \in B : \exists a \in A : a \lesssim b \lesssim c$, from which follows $\forall c \in C : \exists a \in A : a \lesssim c$, as required. □

One might think that there are more natural extensions of the syntactic ordering to the domain of sets than the one given in Definition 5.3. We could, for example, define A less than B if and only if all elements of A are less than all elements of B. Notice, however, that this definition is not a pre-order. Assume that $A = \{a_1, a_2\}$ where $a_1 \not\lesssim\not\gtrsim a_2$. This implies, according to the hypothetical definition above, that $A \lesssim A$ is not true, thus reflexivity is broken. This shows that we must not have an overly strong requirement on comparability. We shall allow some elements to be incomparable so long as there is one element which is comparable. Definition 5.3 captures the right balance in this area, and it is still an effective extension of the syntactic ordering, since if given two one-element sets $A = \{a\}$ and $B = \{b\}$, then $A \lesssim B$ if and only if $a \lesssim b$.

Now that we have all the necessary definitions we can turn to minimal slices. Since minimal slices are not necessarily unique, we shall work with

sets of minimal slices, which are formally defined below.

Definition 5.4 (Set of All Minimal Slices)**.** The set of all minimal slices of a program p for a given projection $(\lesssim, \approx)$, denoted by $\mathbb{M}_p(\lesssim, \approx)$, is defined as follows:

$$\mathbb{M}_p(\lesssim, \approx) = \{q | q \in \mathbb{S}_p(\lesssim, \approx) \text{ and } \nexists q' \in \mathbb{S}_p(\lesssim, \approx) : q' \lesssim_{\not\sim} q\},$$

where $q' \lesssim_{\not\sim} q$ is an abbreviation for $q' \lesssim q \wedge q \not\lesssim q'$.

Below we state the central theorem regarding the connection between the sets of slices and the sets of minimal slices. Informally, given a program, if its slices for projection A are valid slices for projection B as well, then the minimal slices for B are smaller than the minimal slices for A.

Theorem 5.11 (Duality of Slices)**.** *Let $\approx_A$ and $\approx_B$ be semantic equivalences and let $\lesssim$ be such a syntactic ordering that every set of programs has a minimal element with respect to $\lesssim$. Then for any program p the following holds:*

$$\mathbb{S}_p(\lesssim, \approx_A) \subseteq \mathbb{S}_p(\lesssim, \approx_B) \Rightarrow \mathbb{M}_p(\lesssim, \approx_B) \lesssim \mathbb{M}_p(\lesssim, \approx_A).$$

Proof. We need to demonstrate that

$$\forall a \in \mathbb{M}_p(\lesssim, \approx_A) : \exists b \in \mathbb{M}_p(\lesssim, \approx_B) : b \lesssim a.$$

Observe that if $a \in \mathbb{M}_p(\lesssim, \approx_A)$ then, by definition, $a \in \mathbb{S}_p(\lesssim, \approx_A)$, and also $a \in \mathbb{S}_p(\lesssim, \approx_B)$, as $\mathbb{S}_p(\lesssim, \approx_A) \subseteq \mathbb{S}_p(\lesssim, \approx_B)$. If there is no $b \in \mathbb{S}_p(\lesssim, \approx_B)$ such that $b \lesssim_{\not\sim} a$ then $a \in \mathbb{M}_p(\lesssim, \approx_B)$ (by Definition 5.4). Otherwise, $\exists b \in \mathbb{S}_p(\lesssim, \approx_B) : b \lesssim_{\not\sim} a$. Since we require $\lesssim$ to have a minimal element for every set of programs, $\exists b' \in \mathbb{M}_p(\lesssim, \approx_B) : b' \lesssim b$. In either case there is an element of $\mathbb{M}_p(\lesssim, \approx_B)$ which is $\lesssim a$. □

Notice that in the above theorem we added the requirement of having a minimal element in all sets of programs to the syntactic ordering pre-order. Fortunately, this requirement is not overly strict; those syntactic orderings that behave in an intuitive way (i.e., a program is considered smaller only

Program $p_{5.4}$	$q_{5.4}$	$q'_{5.4}$	$q''_{5.4}$
1 x=1; 2 x=2; 3 if (x>1) 4 y=1; 5 else 6 y=1; 7 x=input(); 8 if (x<1) 9 z=0; 10 else 11 z=x*y; 12 w=z;	4 y=1; 7 x=input(); 8 if (x<1) 9 z=0; 10 else 11 z=x*y; 12 w=z;	6 y=1; 7 x=input(); 8 if (x<1) 9 z=0; 10 else 11 z=x*y; 12 w=z;	9 z=0; 12 w=z;

$\sigma_{5.4} = \langle 0 \rangle, V_{5.4} = \{\mathtt{z}\}, n_{5.4} = 12, k_{5.4} = 1$

Figure 5.4: Example program which shows that the reverse of the duality theorem is not true.

if it has fewer statements) fulfil this requirement. The traditional syntactic ordering we use throughout the thesis meets this requirement, and in another example, the amorphous syntactic ordering studied by Harman et al. [49] has the same property.

Interestingly, the converse of Theorem 5.11 does not hold, i.e., $\mathbb{M}_p(\lesssim, \approx_B) \lesssim \mathbb{M}_p(\lesssim, \approx_A)$ does not imply $\mathbb{S}_p(\lesssim, \approx_A) \subseteq \mathbb{S}_p(\lesssim, \approx_B)$. As a counter example, consider program p in Figure 5.4. In this case, there are two minimal static slices, i.e., $\mathbb{M}_{p_{5.4}}(\sqsubseteq, \mathcal{S}(V_{5.4}, n_{5.4})) = \{q_{5.4}, q'_{5.4}\}$, while the set of minimal (Korel-and-Laski-style) dynamic slices consists of just one element, $\mathbb{M}_{p_{5.4}}(\sqsubseteq, \mathcal{D}_{KLi}(\sigma_{5.4}, V_{5.4}, {n_{5.4}}^{(k_{5.4})})) = \{q''_{5.4}\}$. Clearly, $\mathbb{M}_{p_{5.4}}(\sqsubseteq, \mathcal{D}_{KLi}(\sigma_{5.4}, V_{5.4}, {n_{5.4}}^{(k_{5.4})})) \sqsubseteq \mathbb{M}_{p_{5.4}}(\sqsubseteq, \mathcal{S}(V_{5.4}, n_{5.4}))$, since $q''_{5.4} \sqsubseteq q_{5.4}$ and $q''_{5.4} \sqsubseteq q'_{5.4}$, but $\mathbb{S}_{p_{5.4}}(\sqsubseteq, \mathcal{S}(V_{5.4}, n_{5.4})) \not\subseteq \mathbb{S}_{p_{5.4}}(\sqsubseteq, \mathcal{D}_{KLi}(\sigma_{5.4}, V_{5.4}, {n_{5.4}}^{(k_{5.4})}))$, since $q'_{5.4} \notin \mathbb{S}_{p_{5.4}}(\sqsubseteq, \mathcal{D}_{KLi}(\sigma_{5.4}, V_{5.4}, {n_{5.4}}^{(k_{5.4})}))$.

The above theorem provides the basis for a comparison of slicing techniques. It provides the necessary machinery for formalising observations such as 'dynamic slices are smaller than static slices'. This is done for all possible programs and all possible slicing criteria admissible to a chosen form of slicing. To facilitate this formalisation and thus be able to determine whether one definition of slicing leads to inherently smaller slices than another, we

will extend syntactic ordering to apply to *slicing techniques*.

Definition 5.5 (Syntactic Ordering of Slicing Techniques)**.** For any two slicing techniques, $(\lesssim, \approx_A)$ and $(\lesssim, \approx_B)$,

$$(\lesssim, \approx_A) \ \lesssim \ (\lesssim, \approx_B)$$
$$\text{if and only if}$$
$$\forall p, \sigma, V, n, k : \mathbb{M}_p(\lesssim, \approx_A^{(\sigma,V,n,k)}) \ \lesssim \ \mathbb{M}_p(\lesssim, \approx_B^{(\sigma,V,n,k)}).$$

Now we will show that a duality exists between subsumes relation and syntactic ordering over slicing techniques.

Theorem 5.12 (Duality of Slicing Techniques)**.** *For any two slicing techniques* $(\lesssim, \approx_A)$ *and* $(\lesssim, \approx_B)$ *where* $\lesssim$ *is such a syntactic ordering that every set of programs has a minimal element with respect to* $\lesssim$*,*

$$(\lesssim, \approx_A) \ \subseteq \ (\lesssim, \approx_B) \Rightarrow (\lesssim, \approx_B) \ \lesssim \ (\lesssim, \approx_A).$$

Proof. According to Definition 5.2, $(\lesssim, \approx_A) \ \subseteq \ (\lesssim, \approx_B)$ means $\forall p, \sigma, V, n, k : \mathbb{S}_p(\lesssim, \approx_A^{(\sigma,V,n,k)}) \subseteq \mathbb{S}_p(\lesssim, \approx_B^{(\sigma,V,n,k)})$. Theorem 5.11 proved that $\forall p, \sigma, V, n, k : \mathbb{M}_p(\lesssim, \approx_B^{(\sigma,V,n,k)}) \ \lesssim \ \mathbb{M}_p(\lesssim, \approx_A^{(\sigma,V,n,k)})$, which, by Definition 5.5, is equivalent to $(\lesssim, \approx_B) \ \lesssim \ (\lesssim, \approx_A)$. □

This theorem tells us that if slicing technique B subsumes slicing technique A, then the minimal slices of B will be less than those of A. That is, A will tend to produce larger slices.

5.4 Traditional Syntactic Ordering of the Eight Forms of Slicing

Although syntactic ordering in general is only a pre-order, the eight slicing techniques obtained by the combination of the traditional syntactic ordering $\sqsubseteq$ and the eight equivalences (as given in Definition 4.6) result in a lattice isomorphic (in this case inverted) to that given in Figure 5.3. This is shown in Figure 5.5.

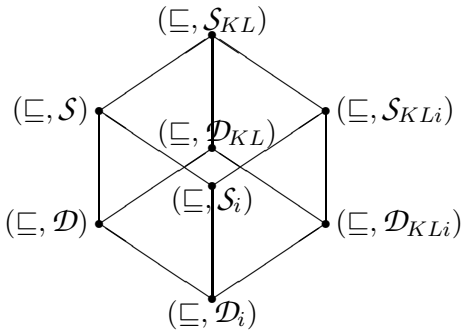


Figure 5.5: Slicing techniques ordered by traditional syntactic ordering.

Theorem 5.12 proves that whenever two slicing techniques are related in Figure 5.3, i.e., they are in subsumes relation, then they have an inverse syntactical ordering relationship. That is, in the "if" direction, the correctness of Figure 5.5 is proven. However, $(\lesssim, \approx_A) \not\sqsubseteq (\lesssim, \approx_B)$ does not imply that $(\lesssim, \approx_B) \not\sqsubseteq (\lesssim, \approx_A)$; thus, it must be shown that the slicing techniques not related in Figure 5.5 are really not related according to the traditional syntactic ordering. This is the role of the following theorem.

Theorem 5.13 (Duality of the Eight Forms of Slicing (only if))**.** *If two slicing techniques are not connected in Figure 5.5, then they are not related according to the traditional syntactic ordering.*

Proof. For each unconnected pair of slicing techniques $(\sqsubseteq, \approx_A)$ and $(\sqsubseteq, \approx_B)$ we have to show that $(\sqsubseteq, \approx_A) \not\sqsupseteq\not\sqsubseteq (\sqsubseteq, \approx_B)$; in other words that

$$\exists p, \sigma, V, n, k : \mathbb{M}_p(\sqsubseteq, \approx_A^{(\sigma,V,n,k)}) \not\sqsubseteq \mathbb{M}_p(\lesssim, \approx_B^{(\sigma,V,n,k)})$$
$$\text{and}$$
$$\exists p', \sigma', V', n', k' : \mathbb{M}_{p'}(\sqsubseteq, \approx_B^{(\sigma',V',n',k')}) \not\sqsubseteq \mathbb{M}_{p'}(\sqsubseteq, \approx_A^{(\sigma',V',n',k')}).$$

The proof makes use of the three counter examples: the one shown in Figure 5.6, and the other two already given in Figures 4.2 and 4.4. The implications of these counter examples are combined to prove the pairs of slicing techniques that go unconnected in Figure 5.5 incomparable.

First, we show that execution path aware (Korel-and-Laski-style) slicing

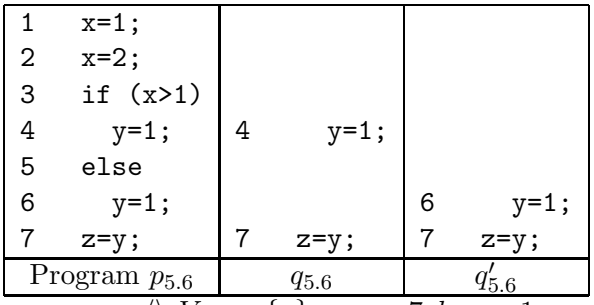


Figure 5.6: Non–KL (execution path unaware) minimal slices.

techniques, denoted by a subscript KL, are not smaller than execution path unaware (or non-Korel-and-Laski-style) ones, denoted by a subscript $\neg KL$. Figure 5.6 gives $p_{5.6}$, $\sigma_{5.6}$, $V_{5.6}$, $n_{5.6}$ and $k_{5.6}$, while the two equations below give the sets of minimal slices for each technique.

$$\begin{aligned} M_{KL} &= \{q_{5.6}\} \\ &= \mathbb{M}_{p_{5.6}}(\sqsubseteq, \mathcal{S}_{KL}(V_{5.6}, n_{5.6})) \\ &= \mathbb{M}_{p_{5.6}}(\sqsubseteq, \mathcal{S}_{KLi}(V_{5.6}, {n_{5.6}}^{(k_{5.6})})) \\ &= \mathbb{M}_{p_{5.6}}(\sqsubseteq, \mathcal{D}_{KL}(\sigma_{5.6}, V_{5.6}, n_{5.6})) \\ &= \mathbb{M}_{p_{5.6}}(\sqsubseteq, \mathcal{D}_{KLi}(\sigma_{5.6}, V_{5.6}, {n_{5.6}}^{(k_{5.6})})) \end{aligned}$$

$$\begin{aligned} M_{\neg KL} &= \{q_{5.6}, q'_{5.6}\} \\ &= \mathbb{M}_{p_{5.6}}(\sqsubseteq, \mathcal{S}(V_{5.6}, n_{5.6})) \\ &= \mathbb{M}_{p_{5.6}}(\sqsubseteq, \mathcal{S}_i(V_{5.6}, {n_{5.6}}^{(k_{5.6})})) \\ &= \mathbb{M}_{p_{5.6}}(\sqsubseteq, \mathcal{D}(\sigma_{5.6}, V_{5.6}, n_{5.6})) \\ &= \mathbb{M}_{p_{5.6}}(\sqsubseteq, \mathcal{D}_i(\sigma_{5.6}, V_{5.6}, {n_{5.6}}^{(k_{5.6})})) \end{aligned}$$

From this it follows by definition that $M_{KL} \not\sqsubseteq M_{\neg KL}$, since $\nexists q \in M_{KL} : q \sqsubseteq q'_{5.6} (\in M_{\neg KL})$.

Now we will prove that iteration count unaware forms of slicing are not smaller than iteration count aware ones. Since $q_{4.2}$ in Figure 4.2 is constructed such that it is a minimal slice of $p_{4.2}$ for the iteration count aware forms of slicing with respect to $\sigma_{4.2} = \langle\rangle$, $V_{4.2}$, $n_{4.2}$, and $k_{4.2}$, we can re-use it here. The two equations below give the minimal slice sets for the eight slicing

techniques.

$$\begin{aligned} M_i &= \{q_{4.2}\} \\ &= \mathbb{M}_{p_{4.2}}(\sqsubseteq, \mathcal{S}_i(V_{4.2}, {n_{4.2}}^{(k_{4.2})})) \\ &= \mathbb{M}_{p_{4.2}}(\sqsubseteq, \mathcal{S}_{KLi}(V_{4.2}, {n_{4.2}}^{(k_{4.2})})) \\ &= \mathbb{M}_{p_{4.2}}(\sqsubseteq, \mathcal{D}_i(\sigma_{4.2}, V_{4.2}, {n_{4.2}}^{(k_{4.2})})) \\ &= \mathbb{M}_{p_{4.2}}(\sqsubseteq, \mathcal{D}_{KLi}(\sigma_{4.2}, V_{4.2}, {n_{4.2}}^{(k_{4.2})})) \end{aligned}$$

$$\begin{aligned} M_{\neg i} &= \{p_{4.2}\} \\ &= \mathbb{M}_{p_{4.2}}(\sqsubseteq, \mathcal{S}(V_{4.2}, n_{4.2})) \\ &= \mathbb{M}_{p_{4.2}}(\sqsubseteq, \mathcal{S}_{KL}(V_{4.2}, n_{4.2})) \\ &= \mathbb{M}_{p_{4.2}}(\sqsubseteq, \mathcal{D}(\sigma_{4.2}, V_{4.2}, n_{4.2})) \\ &= \mathbb{M}_{p_{4.2}}(\sqsubseteq, \mathcal{D}_{KL}(\sigma_{4.2}, V_{4.2}, n_{4.2})) \end{aligned}$$

Again, by definition, the above equations imply that $M_{\neg i} \not\sqsubseteq M_i$, since $q_{4.2}(\in M_i) \sqsubset p_{4.2}(\in M_{\neg i})$.

Finally, we will show that static forms of slicing are not smaller than dynamic forms. In Figure 4.4, $p_{4.4}$, $q_{4.4}$, $\sigma_{4.4}$, $V_{4.4}$, $n_{4.4}$ and $k_{4.4}$ were given and below the sets of minimal slices are listed for the eight slicing techniques.

$$\begin{aligned} M_S &= \{p_{4.4}\} \\ &= \mathbb{M}_{p_{4.4}}(\sqsubseteq, \mathcal{S}(V_{4.4}, n_{4.4})) \\ &= \mathbb{M}_{p_{4.4}}(\sqsubseteq, \mathcal{S}_{KL}(V_{4.4}, n_{4.4})) \\ &= \mathbb{M}_{p_{4.4}}(\sqsubseteq, \mathcal{S}_i(V_{4.4}, {n_{4.4}}^{(k_{4.4})})) \\ &= \mathbb{M}_{p_{4.4}}(\sqsubseteq, \mathcal{S}_{KLi}(V_{4.4}, {n_{4.4}}^{(k_{4.4})})) \end{aligned}$$

$$\begin{aligned} M_D &= \{q_{4.4}\} \\ &= \mathbb{M}_{p_{4.4}}(\sqsubseteq, \mathcal{D}(\sigma_{4.4}, V_{4.4}, n_{4.4})) \\ &= \mathbb{M}_{p_{4.4}}(\sqsubseteq, \mathcal{D}_{KL}(\sigma_{4.4}, V_{4.4}, n_{4.4})) \\ &= \mathbb{M}_{p_{4.4}}(\sqsubseteq, \mathcal{D}_i(\sigma_{4.4}, V_{4.4}, {n_{4.4}}^{(k_{4.4})})) \\ &= \mathbb{M}_{p_{4.4}}(\sqsubseteq, \mathcal{D}_{KLi}(\sigma_{4.4}, V_{4.4}, {n_{4.4}}^{(k_{4.4})})) \end{aligned}$$

The implication of these equations is similar to the above ones, namely $M_S \not\sqsubseteq M_D$, since $q_{4.4}(\in M_D) \sqsubset p_{4.4}(\in M_S)$.

The table below shows how the above three counter examples are used to prove that the slicing techniques unconnected in the lattice in Figure 5.5

are not in relation according to the traditional syntactic ordering.

Incomparability	Follows from
$(\sqsubseteq, \mathcal{D}) \not\supseteq\not\subseteq (\sqsubseteq, \mathcal{D}_{KLi})$	$M_{KL} \not\sqsubseteq M_{\neg KL}$ and $M_{\neg i} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{D}) \not\supseteq\not\subseteq (\sqsubseteq, \mathcal{S}_i)$	$M_S \not\sqsubseteq M_D$ and $M_{\neg i} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{D}_{KLi}) \not\supseteq\not\subseteq (\sqsubseteq, \mathcal{S}_i)$	$M_S \not\sqsubseteq M_D$ and $M_{KL} \not\sqsubseteq M_{\neg KL}$
$(\sqsubseteq, \mathcal{D}_{KL}) \not\supseteq\not\subseteq (\sqsubseteq, \mathcal{S}_i)$	$M_S \not\sqsubseteq M_D$ and $M_{\neg i} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{D}_{KL}) \not\supseteq\not\subseteq (\sqsubseteq, \mathcal{S})$	$M_S \not\sqsubseteq M_D$ and $M_{KL} \not\sqsubseteq M_{\neg KL}$
$(\sqsubseteq, \mathcal{D}_{KL}) \not\supseteq\not\subseteq (\sqsubseteq, \mathcal{S}_{KLi})$	$M_S \not\sqsubseteq M_D$ and $M_{\neg i} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{S}) \not\supseteq\not\subseteq (\sqsubseteq, \mathcal{S}_{KLi})$	$M_{KL} \not\sqsubseteq M_{\neg KL}$ and $M_{\neg i} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{D}) \not\supseteq\not\subseteq (\sqsubseteq, \mathcal{S}_{KLi})$	$M_S \not\sqsubseteq M_D$ and $M_{\neg i} \not\sqsubseteq M_i$
$(\sqsubseteq, \mathcal{D}_{KLi}) \not\supseteq\not\subseteq (\sqsubseteq, \mathcal{S})$	$M_S \not\sqsubseteq M_D$ and $M_{KL} \not\sqsubseteq M_{\neg KL}$

□

Earlier, Theorem 5.12 established the connection between the two fundamental relationships between slicing techniques: subsumption and syntactic ordering. The subsumption relationship tells us when one form of slicing can be used in the place of another, while the syntactic ordering tells us which produces the best (i.e., smallest) slices. Now Theorem 5.13 proves that for the eight forms of slicing we investigated, the lattice of the slicing techniques ordered by the traditional syntactic ordering is isomorphic to that for subsumption (in this case inverted, as a result of duality).

Chapter 6

Conclusions

This part of the thesis presented results concerning the theory of program slicing. The projection theory was used to uncover the precise relationship between various forms of dynamic slicing and static slicing. It had previously been thought that there had been only two nodes in the subsumption relationship between static and dynamic slicing. That is, it was thought that the dynamic slicing criterion merely adds the input sequence to the static criterion and this is all that there is to the difference between the two.

However, the results of the study presented here show that the original dynamic slicing criterion introduced by Korel and Laski contains two additional aspects over and above the input sequence. These are the iteration count and the requirement of maintaining a form of projected path equivalence to the original program. These two additional criteria were shown to be orthogonal components of the original dynamic slicing definition. These two new dimensions can be treated as separate criteria in their own right and may find applications which have yet to be fully exploited by the program slicing community.

The previous sections considered two forms of subsumption relationship. The first is the relationship between the semantic properties of a slice, as captured in the equivalence maintained by slicing. The second relationship concerns the relationship between the slices which may be constructed by the equivalence relations. Thus the first subsumption relationship simply tells us

about the semantic projections denoted by the different forms of the slicing criteria, while the second concerns the slices which may be produced when this semantic requirement is combined with a syntactic ordering. The results make clear that the two lattices so-constructed are isomorphic.

In addition, the syntactic ordering relationship between slicing techniques for static and dynamic slicing was also investigated. It was shown that syntactic ordering is a mirror image of the subsumes relationship, leading to an inverted but isomorphic lattice of inter-technique relationships. The results also tell us that the sets of minimal slices are useful when examining the relationships between slicing techniques.

Part II

Slicing of Binary Programs

Chapter 7

Introduction to Binary Slicing

As described in the previous part of the thesis, program slicing is a technique originally introduced by Weiser [104] for automatically decomposing programs. Since the introduction of the original concept several algorithms have been proposed [87, 60, 49, 69, 21, 66, 9]. These algorithms were originally developed for slicing high-level structured programs, and so, they usually do not handle unstructured control flow correctly and yield imprecise results. Another source of imprecision is the complexity of the static resolution of pointers. Several modifications and improvements have been published to overcome imprecise behaviour [1, 6, 25, 94, 4, 70].

Although lots of papers have appeared in the literature on the slicing of programs written in a high-level language, comparatively little attention has been paid to the slicing of binary executable programs. Cifuentes and Frabuolet [27] presented a technique for the intraprocedural slicing of binary executables, but we are not aware of any usable interprocedural solution. Bergeron et al. [8] suggested using dependence graph-based interprocedural slicing to analyse binaries, but they did not discuss how to handle the problems which arise or provide any concrete experimental results.

The lack of existing solutions is really hard to understand since the application domain for slicing binaries is similar to the one for slicing high-level languages. Furthermore, there are special applications of the slicing of programs without source code like assembly programs, legacy software,

commercial off-the-shelf (COTS) products, viruses and post-link time modified programs. (These include source code recovery, binary bug fixing and code transformation.) Security is also becoming an increasingly important topic: the detection of malicious code fragments is now a major concern of researchers. The slicing of binary executables can be a useful method for helping extract security critical code fragments [8].

Naturally, since the topic of binary slicing is not well covered, difficulties may arise in various parts of the slicing process, especially in the control flow analysis and data dependence analysis of binary executables. These may require special handling techniques.

In this part of the thesis, we will present a method for the interprocedural static slicing of binary executables. First, we will introduce conservative approaches for handling unresolved function calls and branching instructions as well as a safe but imprecise memory model. Then, we will suggest improvements to eliminate useless edges from both the data dependence graph and the call graph.

Chapter 8

Dependence Graph-based Slicing of Binary Executables

8.1 Control Flow Analysis

Many tasks in the area of code analysis, manipulation and maintenance require a control flow graph (CFG). It is also necessary for program slicing to have a CFG of the sliced program as every step in the slicing process depends on it. Building a CFG for a program written in a high-level well-structured programming language like C or Pascal is usually a simple task and only requires syntactical analysis. However, the control flow analysis of a binary executable has a number of associated problems, as we shall see below.

In a binary executable the program is stored as a sequence of bytes. To be able to analyse the control flow of the program, the program itself has to be recovered from its binary form. This requires that the boundaries of the low-level instructions from which the program is constructed be detected. On architectures with *variable length instructions*, the boundaries may not be detected unambiguously. A typical example for this is the Intel platform. Figure 8.1 shows an example byte sequence interpreted in two ways. This highlights the problem that it has to be detected exactly where the decoding of instructions should start from, since even an offset of one byte can and will yield completely false results. On other architectures where *multiple*

Address	Raw	Interpretation 1	Interpretation 2
0x80592b9	0x8b	mov 0x8(%ebp),%eax	
0x80592ba	0x45		inc %ebp
0x80592bb	0x08		or %cl,0x558bf045(%ecx)
0x80592bc	0x89	mov %eax,0xfffffff0(%ebp)	
0x80592bd	0x45		
0x80592be	0xf0		
0x80592bf	0x8b	mov 0xc(%ebp),%edx	
0x80592c0	0x55		
0x80592c1	0x0c		or $0x89,%al
0x80592c2	0x89	mov %edx,0xffffffec(%ebp)	
0x80592c3	0x55		push %ebp
0x80592c4	0xec		in (%dx),%al

Figure 8.1: Two different interpretations of the same sequence of bytes. The raw binary data is decoded to Intel instructions starting from two different addresses.

Address	Raw	Thumb interpretation	ARM interpretation
0x0000006c	0x1c	add r4,r1,#0	stcne p0xc,c0x1,[r12],#0x14
0x0000006d	0x0c		
0x0000006e	0x1c	add r5,r0,#0	
0x0000006f	0x05		
0x00000070	0x68	ldr r0,[r4,#0]	stmvsda r0!,{r11-r14}
0x00000071	0x20		
0x00000072	0x78	ldrb r0,[r0,#0]	
0x00000073	0x00		
0x00000074	0x28	cmp r0,#42	stmcsda r10!,{r2-r4,r12,r14,pc}
0x00000075	0x2a		
0x00000076	0xd0	beq 0xb2	
0x00000077	0x1c		

Figure 8.2: Two different interpretation of the same sequence of bytes. The raw binary data is decoded to two different instruction sets: Thumb and ARM.

instruction sets are supported at the same time, the problem is to determine which instruction set is used at a given point in the code. Figure 8.2 shows an example byte sequence interpreted as Thumb and ARM instructions, both supported by some ARM CPUs. If the binary representation *mixes code and data*, as is typical for most widespread architectures, their separation has to be carried out as well.

After we have identified the instructions of the program, we may begin

to build the graph. First, the *basic blocks* which will constitute the nodes of the CFG need to be determined. For this, *basic block leader* information has to be collected by analysing the instructions. The instructions following branching or function calling instructions, instructions targeted by branching instructions, first instructions of functions and instructions following instruction set switches are called leaders. Instructions between the leaders form the basic blocks of the program, and these blocks are further grouped to represent *functions*. Next, for each function a special node called the *exit node* is created to represent the single exit point of the corresponding function.

The nodes of the CFG are connected by *control flow*, *call* and *return edges* to represent the appropriate possible control transfers during the execution of the program. Control flow edges connect basic blocks in the CFG to represent the possible flow of control within functions. From basic blocks ending in an instruction representing a *return from a function*, control flow edges lead to the exit node. A basic block that ends in an instruction implementing a function call is called a *call site*, while the basic block following it is called the corresponding *return site*. Call edges connect the call sites with nodes representing the called functions, while *return edges* connect the exit nodes of the functions with the corresponding return sites.

The correct detection of the possible control transfers requires a behaviour analysis of machine instructions. Even the high number of instruction types may be hard to cope with, since the types of instructions at the binary level are much more numerous than the types of control structures at the source level, but the hardest problem arises with those control transfer instructions where the *target cannot be determined unambiguously*. In high-level languages, only indirect function calls fall into this category, but on the binary level, intraprocedural control transfer may be represented this way as well. (Such constructs typically arise from compiling switch structures.) To correctly handle these instructions, two new CFG node types have to be introduced, namely the *unknown function* and *unknown block* nodes, which represent the targets of indirect calls and jumps, respectively. For the unknown function, there is only one globally, while every function containing a statically unresolved jump has its own unknown block. These nodes are

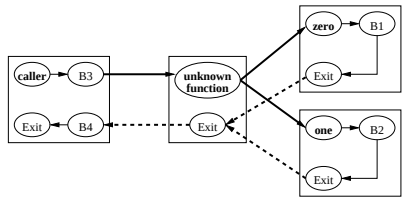
Code			CFG
`zero:`	`mov r0,#0`	*;B1*	
	`mov pc,lr`		
`one:`	`mov r0,#1`	*;B2*	
	`mov pc,lr`		
`caller:`	`stmfd r13!,{r4,lr}`	*;B3*	
	`mov r4,r0`		
	`mov r14,pc`		
	`mov pc,r4` *;indirect call*		
	`ldmfd r13!,{r4,pc}`	*;B4*	

Figure 8.3: An indirect function call with two possible targets in ARM. In the CFG, the thin and thick solid lines represent control flow and call edges, respectively, while the thick dashed lines stand for return edges. The basic block nodes represent the corresponding code fragments, as denoted on the left.

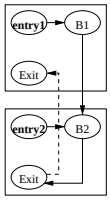
Code			CFG
`entry1:`	`str r0,[r13,#4]`	*;B1*	
	;control falls through		
`entry2:`	`ldr r0,[r13,#8]`	*;B2*	
	`sub r0,#1`		
	`str r0,[r13,#8]`		
	`mov pc,r14`		

Figure 8.4: Two overlapping functions in ARM. In the CFG, the solid lines are control flow edges, while the dashed lines represent compensation control edges. The basic block nodes represent the corresponding code fragments, as denoted on the left.

linked to all the possible targets of the indirect control transfers. Figure 8.3 shows a function which contains an indirect call and two other functions as the possible targets of the call. The corresponding CFG is given as well.

Another type of problems is when control is transferred between functions in a way that is different from a function call. *Overlapping* (or multiple entry) and *cross-jumping* functions, which usually do not occur in high-level languages and result from aggressive interprocedural compiler optimisations, are typical examples of this problem. Since, with these constructs, the exit

```
add:  add r0, r1, r0    ;B1
      mov pc, lr

mul:  push {r4,r5,lr}   ;B2
      add r4, r1, #0
      add r5, r0, #0
      mov r3, #0
      mov r2, #1
      cmp r2, r4
      bgt .l2
.l1:  add r0, r3, #0    ;B3
      add r1, r5, #0
      bl add
      add r3, r0, #0    ;B4
      add r0, r2, #0
      mov r1, #1
      bl add
      add r2, r0, #0    ;B5
      cmp r2, r4
      ble .l1
.l2:  add r0, r3, #0    ;B6
      pop {r4,r5,pc}
```

```
main: push {r4,r5,lr}   ;B7
      add sp, #-8
      bl readin
      add r5, r0, #0    ;B8
      mov r0, #0
      str r0, [sp, #0]
      mov r0, #1
      str r0, [sp, #4]
      mov r4, #1
      cmp r4, r5
      bge .l4
.l3:  ldr r0, [sp, #0]  ;B9
      add r1, r4, #0
      bl add
      str r0, [sp, #0]  ;B10
      ldr r0, [sp, #4]
      bl mul
      str r0, [sp, #4]  ;B11
      add r4, #1
      cmp r4, r5
      blt .l3
.l4:  ldr r0, [sp, #0]  ;B12
      ldr r1, [sp, #4]
      bl writeout
      add sp, #8        ;B13
      pop {r4,r5,pc}
```

Figure 8.5: A Thumb program for computing the sum and product of the first N natural numbers.

node of the control transferring function is not reached, a control flow edge has to be inserted between the exit nodes of the functions to compensate for it. Figure 8.4 gives an example for the overlapping functions and shows their CFG representation.

As the last example in this section, the Thumb assembly listing of the classic slicing example program (which computes the sum and product of the first N natural numbers) is provided in Figure 8.5, while the corresponding CFG is given in Figure 8.6.

During our discussion of the problems above we left open some questions: How might we detect the instruction boundaries? How might we locate instruction set switching points? How should we separate code from data? How might we determine the boundaries of functions? How should we iden-

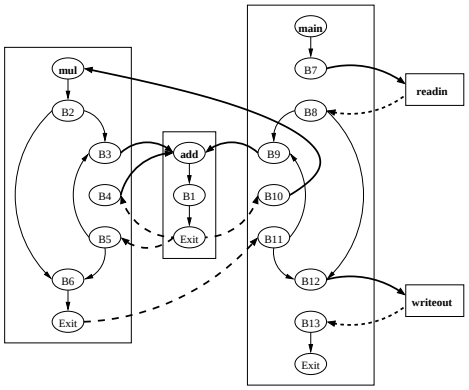


Figure 8.6: The CFG of the program which computes the sum and product of the first N natural numbers. The thin and thick solid lines represent control flow and call edges, respectively, while the thick dashed lines are return edges. The basic block nodes represent the corresponding code fragments, as denoted in Figure 8.5.

tify the potential targets of indirect jumps and calls? It is not possible to furnish a simple and general solution for all these problems, but with some extra information and some architecture specific heuristics the problems may become more manageable. Fortunately, most executable file formats [99, 82] can store extra information along with the raw binary data. The symbolic information, which is usually found in binaries, may be employed to separate code and data in the binary image of the program or assist in detecting function boundaries and instruction set switches. In a similar way, relocation information can be most helpful in determining the targets of indirect function calls and ambiguous control transfers. Usually a hand-written assembly code can also be analysed with no, or very little, extra user input.

Needless to say, the kinds of information stored in the files are highly dependent on the hardware and operating system the binary executable is going to run on, the tool chain the program is generated with, and the file format used. Hence, we cannot say in general how useful data can be extracted from symbolic and relocation information will be. However, our experiences with three kinds of tool chains and file formats on the ARM platform have shown that with an appropriate compiler, file format and architecture specification,

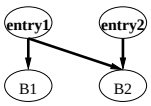


Figure 8.7: The CDG of two overlapping functions in ARM. The thick solid lines denote control dependence. The basic block nodes represent the corresponding code fragments in Figure 8.4.

the necessary information can be retrieved relatively easily.

8.2 Building the Program Dependence Graph

To slice a binary executable, we will perform the following steps: first, we will build an interprocedural control flow graph as described in Section 8.1, then we will perform a control and data dependence analysis for each function found in the CFG, which will result in a program dependence graph (PDG). These PDGs can then be used to compute slices.

One component of the PDG of a function is the control dependence graph (CDG), which represents control dependences between the basic blocks of the function. The CDG is computed in a two-step process: since control dependence in the presence of arbitrary control flow is defined in terms of post-dominance in the CFG, we shall use the algorithm described in [78] to find post-dominators, and then we will build the actual CDG according to Ferrante et al. [40]. The resulting graph will consist of nodes representing the basic blocks and function entries, and *control dependence edges* connecting these nodes.

One peculiar feature of binary programs is that the instructions may belong to multiple functions due to overlapping and cross-jumping, a situation that rarely occurs in high-level structured programming languages. This leads to instructions that may depend on multiple function entry nodes. Figure 8.7 shows the CDG of the two overlapping functions presented in Figure 8.4.

The other part of a PDG is the data dependence graph (DDG) representing the dependences between instructions according to their *used* and *defined*

arguments. In high-level languages, the arguments of statements are usually local variables, global variables or formal parameters, but such constructs are generally not present at the binary level. Low-level instructions read and write registers, flags (one bit units) and memory addresses, hence existing approaches have to be adapted to use the appropriate terms.

In our approach, we analyse each instruction in the program and determine which registers and flags it reads and writes. The analysis does not have to take into account the register which controls the flow of the program (usually called the instruction pointer or the program counter), since the effect of this register is captured by the CFG and CDG. However, the memory access of the instructions has to be analysed. A conservative approach is to just find out whether an instruction reads from or writes to the memory. Thus, the whole memory here is represented as a single argument of the instructions. A possible optimisation of this rather conservative approach is discussed in Section 9.1.

The analysis results in the sets u_j and d_j for each instruction j, which contain all used and defined arguments of j, respectively. During the analysis we also determine the sets u_j^a for every $a \in d_j$ which contain the arguments of j actually used to compute the value of a. Obviously $u_j = \bigcup_{a \in d_j} u_j^a$ for each instruction j, but instructions may exist where $u_j^a \subset u_j$ for a defined argument a. High-level programming languages may also have such statements, but usually they can be divided into subexpressions with only one defined argument, which cannot be done with low-level instructions.

Unlike in high-level programs, the parameter list of procedures is not explicitly defined in binaries but has to be determined via a suitable interprocedural analysis. We use a fix-point iteration to collect the sets of *input* and *output parameters* of each function. We compute the sets U_f and D_f (similar to the sets GREF(f) and GMOD(f) in [60]) representing the used and defined arguments of every instruction in function f itself and in the functions called (transitively) from f, as given in Figure 8.8. I_f is the set of instructions in f and C_f is the set of functions called from f. The resulting set D_f is called the set of output parameters of function f, while $U_f \cup D_f$ yields the set of input parameters of f.

$$
\begin{array}{rcl}
U_f^{(0)} & = & \emptyset \\
U_f^{(i+1)} & = & \left(\bigcup_{j \in I_f} u_j\right) \cup \left(\bigcup_{g \in C_f} U_g^{(i)}\right) \\
U_f & = & U_f^{(i)}, \text{ where } U_f^{(i)} = U_f^{(i+1)}
\end{array}
$$

$$
\begin{array}{rcl}
D_f^{(0)} & = & \emptyset \\
D_f^{(i+1)} & = & \left(\bigcup_{j \in I_f} d_j\right) \cup \left(\bigcup_{g \in C_f} D_g^{(i)}\right) \\
D_f & = & D_f^{(i)}, \text{ where } D_f^{(i)} = D_f^{(i+1)}
\end{array}
$$

Figure 8.8: U_f and D_f

$$C_{\text{add}} = \emptyset, C_{\text{mul}} = \{\text{add}\}, C_{\text{main}} = \{\text{add}, \text{mul}, \text{readin}, \text{writeout}\}$$
$$U_{\text{readin}} = U_{\text{writeout}} = D_{\text{readin}} = D_{\text{writeout}} = \{\text{R0} - \text{R12}, \text{SP}, \text{LR}, \text{mem}\}$$

$$
\begin{array}{rcl@{\qquad}rcl}
U_{\text{add}}^{(0)} & = & \{\text{R0}, \text{R1}, \text{LR}\} & D_{\text{add}}^{(0)} & = & \{\text{R0}\} \\
U_{\text{mul}}^{(0)} & = & \{\text{R0} - \text{R5}, \text{SP}, \text{LR}, \text{mem}\} & D_{\text{mul}}^{(0)} & = & \{\text{R0} - \text{R5}, \text{SP}, \text{LR}, \text{mem}\} \\
U_{\text{main}}^{(0)} & = & \{\text{R0}, \text{R4}, \text{R5}, \text{SP}, \text{LR}, \text{mem}\} & D_{\text{main}}^{(0)} & = & \{\text{R0}, \text{R1}, \text{R4}, \text{R5}, \text{SP}, \text{LR}, \text{mem}\} \\
U_{\text{add}}^{(1)} & = & U_{\text{add}}^{(0)} & D_{\text{add}}^{(1)} & = & D_{\text{add}}^{(0)} \\
U_{\text{mul}}^{(1)} & = & U_{\text{mul}}^{(0)} & D_{\text{mul}}^{(1)} & = & D_{\text{mul}}^{(0)} \\
U_{\text{main}}^{(1)} & = & \{\text{R0} - \text{R5}, \text{SP}, \text{LR}, \text{mem}\} & D_{\text{main}}^{(1)} & = & \{\text{R0} - \text{R5}, \text{SP}, \text{LR}, \text{mem}\} \\
U_{\text{add}}^{(2)} & = & U_{\text{add}}^{(1)} & D_{\text{add}}^{(2)} & = & D_{\text{add}}^{(1)} \\
U_{\text{mul}}^{(2)} & = & U_{\text{mul}}^{(1)} & D_{\text{mul}}^{(2)} & = & D_{\text{mul}}^{(1)} \\
U_{\text{main}}^{(2)} & = & U_{\text{main}}^{(1)} & D_{\text{main}}^{(2)} & = & D_{\text{main}}^{(1)}
\end{array}
$$

Figure 8.9: Computing U_f and D_f sets for the functions of the program given in Figure 8.5.

Figure 8.9 shows the evaluation of the U_f and D_f sets for the functions of the example program in Figure 8.5. The iteration for functions `readin` and `writeout` is not detailed, but the fixpoint is given.

Using the results of the above analyses we extend the CDG with appropriate nodes to form the basis of the DDG. We insert nodes into the graph to represent the instructions of the program, where, of course, each depends on its basic block. We also insert nodes to represent the used and defined arguments of each instruction; these nodes are in turn dependent on the corresponding instructions. Next, for basic blocks, which act as call sites, we add control dependent nodes representing the parameters of the called function. *Actual-in* and *actual-out* parameter nodes are created for the input and output parameters of the called function, respectively. Finally, for

the function entry nodes we add control dependent *formal-in* and *formal-out* parameter nodes to represent the formal input and output parameters of the functions.

Once the appropriate nodes have been inserted, the data dependence edges are added to the graph. First, we add the data dependence edges which represent a dependence inside individual instructions: the definition of argument a in instruction j is data dependent on the use of argument a' in j if $a' \in u_j^a$. Then, the data dependences between instructions are analysed: the use of argument a in instruction j depends on the definition of a in instruction k if definition of a in k is a *reaching definition* for the use of a in j, which means that there exists a path in the CFG from k to j such that a is not redefined. The above definition for the notion of reaching definition is suitable for flags and registers, but it has to be relaxed for memory access. The definition of memory in an instruction k is a reaching definition for the use of memory in another instruction j, if there is a path in the CFG from k to j, even if there is another instruction on that path which defines memory, since the whole memory is represented as a single argument. In our analysis, the call site basic blocks are viewed as pseudo instructions which are placed after the last instruction in the block, with actual-in and actual-out parameters treated as used and defined arguments, respectively. Similarily, formal-in and formal-out parameter nodes are treated as defined and used arguments of pseudo instructions at the entry and exit points of functions.

The PDG constructed so far still lacks some dependence edges. Control dependence edges connect basic block nodes, but the dependences are in fact caused by the branching instructions in the blocks. Unfortunately, these dependeces are not represented in the PDG in its current form. Therefore, to make it precise, the PDG has to be augmented with additional control dependence edges for compensation.

As an example for a fully built PDG, Figure 8.10 shows the PDG of function `mul` of the example program of Figure 8.5.

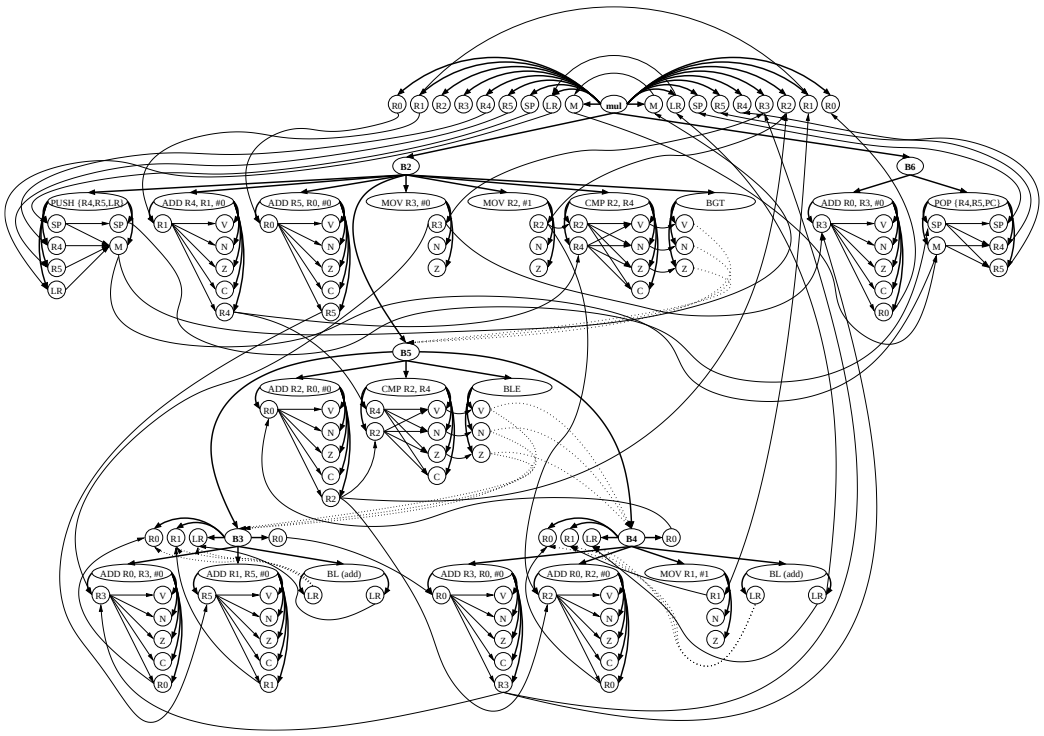


Figure 8.10: The PDG of function `mul` of the example program presented in Figure 8.5. The thick and thin solid lines represent control and data dependence edges respectively, while the dotted lines are compensation control dependence edges. The Rx, LR and SP arguments represent registers, V, N, Z, and C are flags and M stands for the memory.

8.3 Interprocedural Slicing using the System Depencence Graph

The PDGs built so far can be used to compute intraprocedural slices by treating call sites as instructions with actual-in and actual-out parameters as used and defined arguments, respectively, where each defined argument is data dependent on all used arguments. A slice can be computed for any set of used or defined arguments as the slice criterion by traversing via control and data dependence edges [87]. The resulting program slice consists of the instruction nodes reached during the graph traversal.

However, to compute interprocedural slices, the individual PDGs of func-

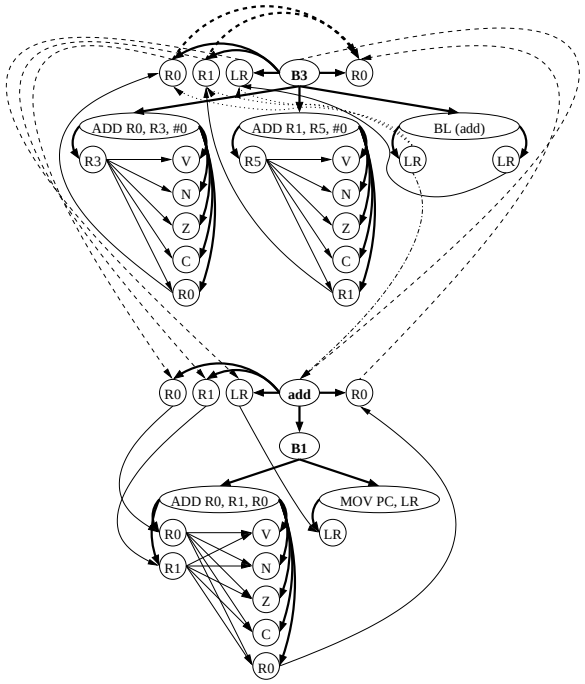


Figure 8.11: A portion of the SDG of the example program given in Figure 8.5 showing the function `add` and call site B3 in the function `mul`. The thick and thin solid lines represent control and data dependence edges, respectively, the thick dashed lines are summary edges, the thin ones denote parameter and call edges, while the dotted and dot-dashed vectors are for control and call compensation edges.

tions need to be interconnected. We connect all actual-in and actual-out parameter nodes with the appropriate formal-in and formal-out nodes using *parameter-in* and *parameter-out* edges to represent parameter passing. We also add summary edges to represent dependences between actual-in and actual-out parameters, see [90]. The resulting graph is the system dependence graph (SDG) of the program.

Similar to the control dependences described in Section 8.2, even though call edges connect basic block and function entry nodes, the real dependences come from the call instructions. To avoid missing dependences, the SDG needs to be augmented with new compensation dependence edges connecting the used arguments of function call instructions to the called function entry

nodes. Figure 8.11 presents a small portion of the SDG of the example program containing a call site and the entry point of the corresponding called function.

The SDG built by using the approach described in this section and in the preceding ones can be used to compute interprocedural slices using the two-pass algorithm of Horwitz et al. [60] with respect to a set of argument nodes. However, experience shows that the computed static slices tend to be quite large. The reason for this will be investigated in the next chapter, where solutions are proposed as well.

Chapter 9

Improving the Slicing of Binary Executables

9.1 Refining Static Analyses

Although the program and system dependence graphs built as described in the previous chapter are safe, they are overly conservative. One reason for this is the conservative approach of the data dependence analysis and the lack of use of architecture specific information. In this section, we will present two approaches for improving the precision of the DDG. One is based on a heuristical analysis of function prologs and epilogs, while the other is a more sophisticated analysis of the memory access of the instructions.

On most current architectures, various function calling conventions exist which specify what portions of the register file a function has to keep intact when called. Functions conforming to such calling conventions usually save registers somewhere to the memory on entry (mostly to the stack) and restore them just before exiting. These register save and restore operations are usually easy to detect by using knowledge of the architecture and the calling convention.

If the set of saved and restored registers can be determined, we can redefine the set of output parameters for a function f as $D_f \setminus S_f$ where D_f is defined in Figure 8.8, and S_f is the set of registers saved on entry and

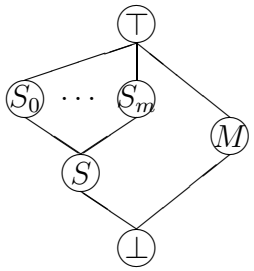


Figure 9.1: The lattice to characterise register content.

restored on exit in f. Using the new set of input and output parameters to build the PDG, the slice is going to suffer less from the imprecision caused by the conservative handling of function calls.

Another source of imprecision is the handling of memory accesses in data dependence analysis. At the binary level, the high-level concepts of variables and function parameters do not exist, so compilers use registers in their place. But in most architectures the number of available registers is limited, and registers are also used to store the temporary results of computations in the program. The parameters and variables that cannot be assigned to registers are usually stored in a specific portion of the memory called the stack.

Since the memory model outlined in Section 8.2 is very simple, a data dependence analysis cannot accurately detect the dependences across the stack, hence the computed slices are too conservative. As a solution to this problem, we propose an improved memory model (a modified data dependence analysis and a propagation algorithm) to aid the analysis.

In our procedure, we characterise all registers at a given instruction location with a pair of lattice elements to represent statically collected information about their contents at the entry and exit points of the instruction. The lattice and its elements are shown in Figure 9.1.

Assigning $\top$ to a register means that it may contain a reference to an (as yet) undetermined stack position. The lattice element $\bot$ tells us that whether the register contains a reference in the stack or not cannot be statically determined. If it is dereferenced it may access not only a stack element but also a memory location outside the stack. Assigning M to a register means

$$
\begin{aligned}
\text{any} \sqcap \top &= \text{any} \\
\text{any} \sqcap \bot &= \bot \\
S_i \sqcap S_j &= S_i \text{ if } i = j \\
S_i \sqcap S_j &= S \text{ if } i \neq j \\
S_i \sqcap S &= S \\
S_i \sqcap M &= \bot \\
S \sqcap M &= \bot
\end{aligned}
$$

Figure 9.2: Rules for $\sqcap$.

that it may not contain a reference in the stack. The lattice element S shows that the register definitely contains a reference somewhere in the stack, but the exact location cannot be determined. Assigning S_i to a register means that the register contains a reference to a known stack element.

For each function, our algorithm starts by assigning $\top$ to all registers both at the entry and exit points of each instruction, except the first one. At the entry point of the first instruction, the algorithm assigns $\bot$ to all registers except the one that specifies the current top of the stack (usually called the stack pointer or SP), which is assigned a value of S_0.

The algorithm uses the CFG to propagate information. First of all, the first instruction is placed on a worklist. Then, iteratively, a node is chosen and removed from the worklist, and it is examined. The lattice elements associated with the registers at the entry of the examined instruction become the meet of the lattice elements associated with the corresponding registers at the exit of the preceeding instructions. The meet rules are given in Figure 9.2. The instruction is evaluated by simulating its behaviour on the new input values, and then the exit values are determined. If any of the computed exit values differ from the corresponding lattice elements associated with the registers at the exit point of the instruction, the instructions following it according to the control flow are added to the worklist. The process is repeated until the worklist is empty.

Using the results of this process, data dependence analysis described in

Section 8.2 can be improved so as to avoid adding superfluous dependence edges to the graph. In the conservative approach, the entire memory was represented by only one argument, but by using the results of the above algorithm, the used and defined arguments can be determined more precisely. Instead of the argument representing the whole memory, we can use arguments labeled according to the same lattice elements as those used in the analysis. This is used to represent certain parts of the memory.

In our current approach, the improved handling of the memory is not applied to formal and actual parameters owing to the difficulties of interprocedural analysis of stack and memory access. The formal and actual parameters represent memory access with a $\perp$ type node.

For the data dependence analysis to make use of the above analysis, the reaching definition has to be modified for arguments representing access to the memory. The definition of the argument a' in instruction k is a reaching definition for the used argument a of instruction j if $a' \sqcap a \in \{a, a'\}$ and there is a path in the CFG from k to j such that if a' is some S_i, then a' is not redefined.

Figure 9.3 shows the interprocedural backward slice of the example program given in Figure 8.5 with respect to R0 used by the instruction at label `.14` using the results of the here-presented optimisations. The slice contains the instructions responsible for the loop control logic and the computation of the sum in function `main` and the whole function `add`. Without the improvements, the slice would contain the whole function `mul` and all the instructions in basic blocks B8, B10 and B11.

9.2 Improving the Call Graph with Dynamic Information

Our investigations revealed that the high number of unresolved indirect function calls is another reason why the slices are too large. Antoniol et al. [4] experimented with *static* points-to analysis of C programs, focusing on function pointers and their effect on the call graph. Although their results are

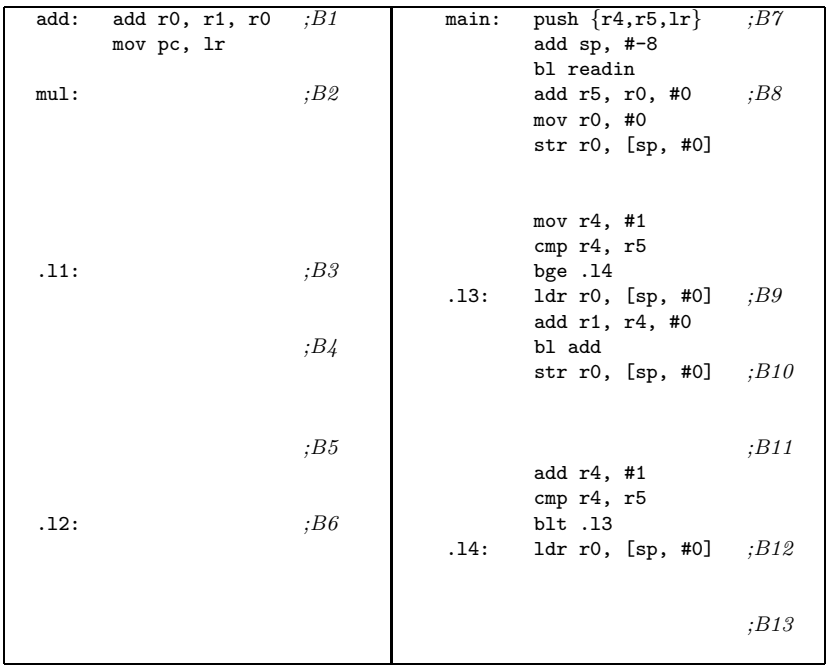


Figure 9.3: An interprocedural backward slice of the example program given in Figure 8.5 w.r.t. R0 used by the instruction at label `.l4`.

impressive, the application of a static points-to analysis with a cubic worst-case complexity on low-level programs where even local variables reside on the stack and are accessed via pointers would prove quite ineffective.

In contrast, Mock et al. [83] examined the effects of using *dynamic* points-to information in the slicing of C programs. Their results suggest that attention should be paid to function pointers and calls through them. Furthermore, since points-to data is collected only for function pointers, the slowdown caused by this type of profiling is minimal, which makes the approach feasible in practice.

In the case of binary executables, the equivalent terms for *function pointer* and *call via a function pointer* are *register* and *statically unresolved indirect call site*, respectively, while *points-to sets* translate to *sets of memory addresses* (of functions) in this context. That is why we are interested in the values of registers at specific call sites.

To facilitate the gathering of dynamic information, we need to determine

the run-time address of each statically unresolved indirect call site after the construction of the CFG is completed. The application can then be executed in a controlled environment on some representative input. These previously determined addresses are used as breakpoints where dumping the registers to a log file should be performed.

The controlled environment could be either a software emulator or real hardware with a debugger interface. We should note, however, that the required information (i.e., the contents of registers at call sites) could be obtained by other means as well – for example, by instrumentation. The drawback of instrumentation is that it requires the modification of the binary code, which is prone to error and must be done with extreme care.

With the help of the generated log files, it is possible to determine the realised targets of the statically unresolved indirect call sites and thus, replace call edges to the unknown function node with call edges to the actual targets. The call sites which were not executed during any invocation of the application have no associated dynamic information with them, so they may be handled in various ways. One alternative is to rely entirely on dynamic data and treat them as calling no functions (which makes them equivalent with no-operation instructions), but this solution may result in over-optimistic slices. The other alternative is to retain the call edge to the unknown function node at these call sites as a fallback. In Section 10.2, we provide results for both approaches. Although the resulting call graphs may be imprecise in both cases, so the slices may become unsafe, in some situations (e.g., when debugging with limited resources) this limitation is acceptable.

Figure 9.4 shows the call graph of an example program named `decode`, while Figure 9.5 shows the same call graph made more precise with the dynamically gathered information at 80% coverage level (unexecuted indirect call sites are treated as no-operation instructions). As is readily apparent from the difference between the two figures, the use of dynamic information can result in a huge reduction in the number of call edges.

Once the dynamic information is processed, only the summary edges in the SDG need to be recomputed, and then the interprocedural static slice can be computed in the usual way.

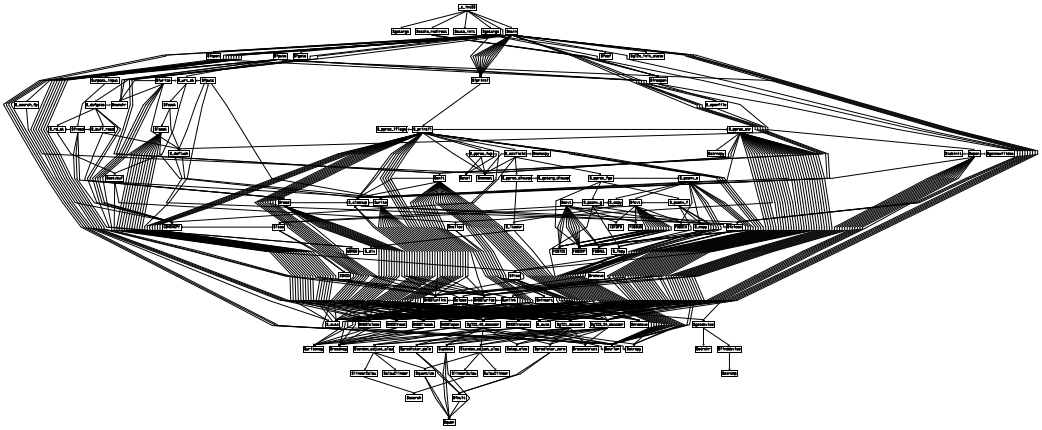


Figure 9.4: The statically computed call graph of the program `decode`.

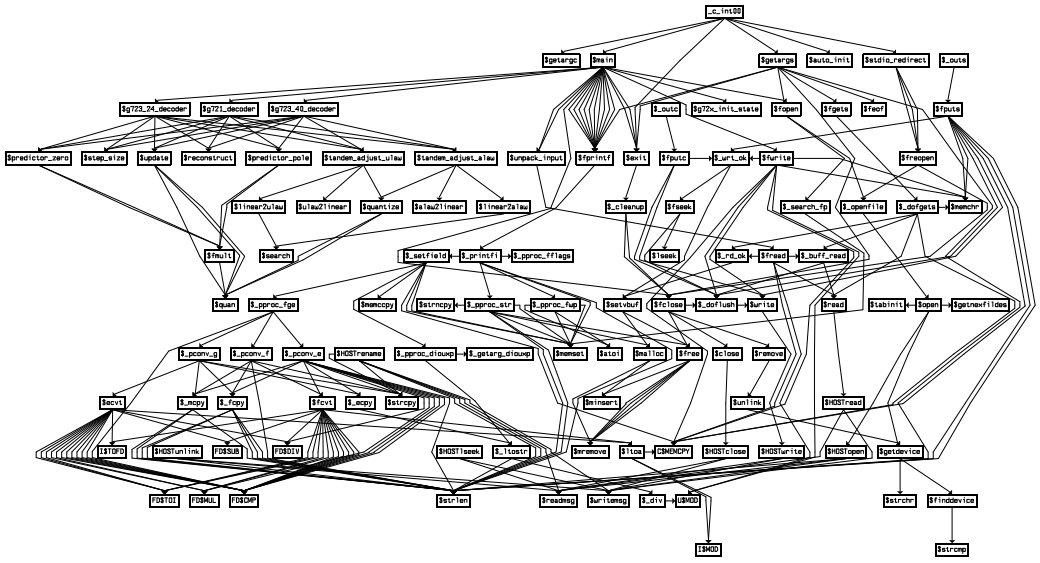


Figure 9.5: The call graph shown in Figure 9.4 made more precise with the help of dynamically gathered information.

Chapter 10

Experimental Results

10.1 Static Slicing

We implemented a slicer for statically linked binary ARM executables and evaluated it on programs taken from the SPEC CINT2000 [96] and MediaBench benchmark suites [76]. The selected programs were compiled using Texas Instruments' TMS470R1x Optimizing C Compiler version 1.27e for the ARM7T processor core with Thumb instruction set. The size of code in the executables ranged from 12 to 419 kilobytes. In Table 10.1, we also state the number of lines in the C source files of the programs and the number of low-level instructions compiled from these sources. The number of instructions originating from the linked libraries is given in parentheses.

First, we built the CFG for all the selected programs, as described in

Table 10.1: The benchmark used to evaluate binary slicing.

Program	**Source lines**	**Raw size**	**Instructions**
ansi2knr	693	12596	774 (+5014)
decode	1593	15476	2074 (+5162)
bzip2	4247	43324	8788 (+5311)
toast	5997	37748	10662 (+5517)
sed	12241	42328	13284 (+5820)
cjpeg	28720	99352	37019 (+7482)
osdemo	62374	419032	177214 (+7025)

Table 10.2: All functions present in the programs and the statically reachable ones. Library functions are given in parentheses for each case.

Program	**All functions**	**Reachable functions**
ansi2knr	5 (+114)	5 (+97)
decode	26 (+109)	21 (+91)
bzip2	73 (+119)	51 (+102)
toast	86 (+124)	72 (+107)
sed	102 (+142)	92 (+128)
cjpeg	381 (+149)	327 (+133)
osdemo	1186 (+145)	675 (+127)

Section 8.1. As one result of this analysis, in Table 10.2 we list for each program the number of functions present in the binary code and the number of functions statically reachable from the entry point of the program according to the statically computed call graph. The first number in each column represents the functions actually present in the sources, while the second one in parentheses stands for library functions. The difference between the number of all functions present in the binary code and the number of reachable functions reveals the inefficiency of the linking process.

Once the CFGs were present, we performed control and data dependence analyses (both the conservative and statically improved ones, as described in Sections 8.2 and 9.1) to obtain PDGs for each reachable function, and finally, we created the SDGs. Table 10.3 shows the summary of edge types in the dependence graphs as well as the effect of the static improvements. As can be seen, the reduction in the number of data dependence and summary edges in the SDGs are, on average 28% and 51%, respectively, with maximum improvements as high as 44% and 58%, respectively.

After obtaining the SDGs for all the benchmark programs, we computed interprocedural slices using both the conservative and the statically improved dependence graphs. To avoid bias from applying a given selection strategy, we decided to compute slices for each instruction of those reachable functions that were compiled from the sources (not added during the linking process). During slicing, we considered the used arguments of the instructions as slice criteria. Table 10.4 shows the results of the computations. We obtained slices

Table 10.3: Summary of edges in the SDGs built using both conservative and improved data dependence analyses.

Program	Control dependence	Data dependence		Summary	
		conservative	improved	conservative	improved
ansi2knr	1953	51535	40181	19859	9626
decode	2018	56650	45253	22864	11099
bzip2	4154	169934	131410	61404	30363
toast	3435	155867	110194	48061	24500
sed	7254	797140	448998	69924	37082
cjpeg	10305	539978	395443	228210	118127
osdemo	48302	4614767	3071640	779437	327126

Table 10.4: Average number of instructions in interprocedural static slices based on both conservative and improved data dependence analyses (no criteria from library code). The contribution of library code to the slice size is given in parentheses.

Program	Criteria	Conservative	Improved
ansi2knr	774	495 (+3145)	486 (+2911)
decode	1670	1205 (+3234)	1167 (+2992)
bzip2	8591	3147 (+3178)	3099 (+2960)
toast	7876	5783 (+3486)	5660 (+3268)
sed	13109	7532 (+3868)	7435 (+3629)
cjpeg	33048	26338 (+5447)	25556 (+5144)
osdemo	141686	97680 (+4262)	95311 (+4004)

that on average had 36%-71% of the source-originated instructions using the conservative approach and 1%-3% fewer instructions with the help of the improvements.

The above results mainly relate to that part of the application which was compiled from sources, but there are situations (e.g., programs modified at post-link time) where library code also becomes important and the entire binary executable needs to be analysed. For this reason in Table 10.5, we list the results on the slices computed for those (reachable) functions which originate from library code as well. The figures in this table show similar trends to the ones in the previous table. The slices computed using the

Table 10.5: Average number of instructions in interprocedural static slices based on both conservative and improved data dependence analyses (criteria taken from all reachable functions).

Program	Criteria	Conservative	Improved
ansi2knr	5137	3754	3520
decode	6177	4457	4190
bzip2	13275	7399	7114
toast	12757	9418	9066
sed	18437	11969	11623
cjpeg	40077	31606	30529
osdemo	148011	102324	99683

conservative dependence graphs on average contained 52%-71% of all the instructions, while the static improvements brought only a 1%-4% decrease in these values. According to our investigations, a key factor in the moderate improvement in the size of interprocedural slices is the high number of statically unresolved function calls.

10.2 The Effect of Dynamic Information

To gather dynamic information about the selected benchmark programs, we executed them in the emulator of Texas Instruments' TMS470R1x C Source Debugger. We used the test inputs that came with the benchmark suites to achieve as good a code coverage as possible. In Table 10.6 we list for each program the number of functions called during the executions of the program. The number of executed functions shows, when compared to the figures given in Table 10.2, that in some cases it is possible to get a very good code coverage using the default test inputs, but in others – especially in the case of larger programs – the achieved coverage ratio is poor. However, sometimes – e.g. in the case of `osdemo` – it is impossible to achieve a better ratio, since a big portion of the code turns out to be never executed, even though static analysis marks it as reachable. This is usually the case when the number of indirectly called functions is high. After all, unless all realisable paths of execution can be covered by the test inputs, the call graph resulting

Table 10.6: The functions called during test executions. Library functions are given in parentheses for each case.

Program	Executed functions
ansi2knr	4 (+57)
decode	21 (+45)
bzip2	38 (+53)
toast	60 (+54)
sed	57 (+70)
cjpeg	134 (+62)
osdemo	122 (+85)

from the static call graph improved by dynamically gathered information can be considered only as an approximation of the real, precise call graph.

With the help of the dynamic information collected, we were able to make the call graph at the indirect call sites and their targets more accurate. In Table 10.7, we give the number of indirect call sites and the number of indirectly callable functions for every benchmark program (separately for the application and library parts of the program), while Table 10.8 shows how the number of call edges changes with the improvements, giving results for both approaches handling unexecuted indirect call sites (as described in Section 9.2). Here, the number of indirect call edges is given in parentheses. As expected, the number of call edges is significantly reduced in those applications which make intensive use of indirect function calls (`cjpeg`, `osdemo`). Even those programs that contained only a few indirect call sites and indirectly callable functions showed a clear reduction. However, as a consequence of the poor indirect call site coverage, the reduction becomes only moderate if the static fallback is used, i.e., when the unexecuted call sites call all the possible targets.

To measure the effect of a more precise call graph, we computed slices for the same slicing criteria using the static call graph and the two kinds of dynamically improved ones. Again, to avoid bias from applying a given selection strategy, we computed slices for each instruction of those source-originated functions that were called during the executions of the benchmark programs. In Table 10.9, we list the average number of instructions in the

Table 10.7: Indirect function call sites and indirectly callable functions. Call sites and targets in library code are given in parentheses.

Program	Call sites	Targets
ansi2knr	0 (+12)	0 (+12)
decode	1 (+12)	3 (+10)
bzip2	0 (+12)	0 (+10)
toast	6 (+12)	13 (+12)
sed	1 (+12)	3 (+16)
cjpeg	469 (+32)	181 (+15)
osdemo	245 (+12)	359 (+14)

Table 10.8: Change in the number of call edges as a result of the use of dynamic information. The number of indirect call edges is given in parentheses.

Program	Static	Dynamic with fallback	Dynamic
ansi2knr	401 (144)	324 (67)	264 (7)
decode	428 (169)	358 (99)	267 (8)
bzip2	781 (120)	736 (75)	666 (5)
toast	1010 (450)	774 (214)	574 (14)
sed	1047 (247)	886 (86)	810 (10)
cjpeg	99320 (98196)	83145 (82021)	1217 (93)
osdemo	106819 (95861)	91940 (80982)	10999 (41)

computed slices. The contribution of library code to the slice size is shown in parentheses. The results reveal that there is a high correlation between the reduction of the call edges and the reduction of the size of the slices. In the case of `cjpeg` and `osdemo`, the average size of the slices computed using the dynamically improved call graph fell by 72% and 57%, respectively, compared to the static approach. Two programs using indirect function calls only rarely (`decode` and `toast`) achieved a 6% reduction, but the others, not surprisingly, brought no improvements. In the case of the dynamically improved call graph using the static fallback, the high number of remaining call edges cancelled out nearly all the improvements.

Since, as mentioned in the previous section, there are cases when the entire binary is important, in Table 10.10 we list the results on the slices computed for those functions which originate from library code as well. As

Table 10.9: Change in the average size of slices as a result of using dynamic information (no criteria from library code). The contribution of library code to the slice size is given in parentheses.

Program	Criteria	Static	Dynamic with fallback	Dynamic
ansi2knr	761	485 (+2911)	485 (+2904)	485 (+2782)
decode	1670	1167 (+2992)	1166 (+2984)	1097 (+2719)
bzip2	8237	3084 (+2985)	3084 (+2977)	3084 (+2764)
toast	7428	5695 (+3287)	5693 (+3277)	5373 (+3107)
sed	11218	7282 (+3619)	7281 (+3604)	7159 (+3397)
cjpeg	12103	24038 (+4873)	24003 (+4865)	6700 (+4246)
osdemo	20142	91855 (+3890)	91825 (+3878)	39368 (+3440)

Table 10.10: Change in the average size of slices as a result of using dynamic information (criteria taken from the entire executable set).

Program	Criteria	Static	Dynamic with fallback	Dynamic
ansi2knr	2894	3366	3358	3228
decode	3445	4119	4111	3674
bzip2	10122	6431	6423	6211
toast	9631	8959	8947	8344
sed	14431	11307	11292	10981
cjpeg	14640	28911	28870	11127
osdemo	24319	97890	97850	48514

these figures are similar to the ones shown in Table 10.9, we may draw a similar conclusion as before. With the applications which make extensive use of indirect function calls the dynamically improved call graph can result in a high slice size reduction, but otherwise the improvements are only moderate. Moreover, unless the coverage of the indirect call sites can be greatly improved, there is not much sense in using the improved call graph with the static fallback method.

Chapter 11

Conclusions

In this part of the thesis, we described how interprocedural slicing can be applied to binary executables. First, we discussed the problems associated with the control flow analysis of binary programs, and then we presented a conservative dependence graph-based slicing approach as well as its improvements. First, we experimented with two static improvements, and then we described how superfluous edges could be removed from the statically computed call graph with the help of dynamically gathered information.

We evaluated both approaches on programs compiled for ARM architecture with the help of a prototype implementation of the described methods. Using the conservative approach, we achieved an interprocedural slice size of 52%-71% on average and 1%-4% reduction using the static improvements. The moderate improvements are due to the conservative handling of memory and indirect function calls. The experiments with the dynamic approach demonstrated that the slice size could be dramatically further reduced if the analysed application made extensive use of indirect function calls. The drawback of the method described is that the improved call graph may be unsafe, but the safe call graph can be approximated if sufficient code coverage can be achieved. The resulting call graph may work well in situations where the safety of slices is not critical, e.g. in some debugging scenarios.

Part III

Code Obfuscation via Control Flow Flattening

Chapter 12

Introduction to Code Obfuscation

Protecting programs from unauthorised access has always been a concern of software vendors. Unfortunately, it is impossible to guarantee complete safety, since given sufficient time, any code can be broken. Thus, the goal is usually to make the job of the attacker as difficult as possible. Therefore, several techniques have been suggested to hinder the comprehension and modification of programs.

Some of the protection methods rely on hardware support [109], while others are pure software solutions [32]. Some techniques are static, i.e., they are applied to programs using compile or build time information only, while others protect the code even during runtime, i.e., dynamically [44]. In this part of the thesis, we focus on code obfuscation, which is a first line of defence in the protection of programs, since its goal is to prevent attackers from comprehending the code. If an attacker cannot comprehend the code, he cannot modify it either.

Several code obfuscation techniques exist. Their common feature is that they change programs to make their comprehension difficult, while preserving their original behaviour. The simplest technique is layout transformation [79], which scrambles identifiers in the code, removes comments and debug information. Another technique is data obfuscation [31], which changes

data structures, e.g., by changing variable visibilities or by reordering and restructuring arrays. The third group consists of control flow transformation algorithms where the goal is to hide the control flow of a program from analysers. These algorithms change the predicates of control structures to an equivalent, but more complex code, insert irrelevant statements, or "flatten" the control flow [101, 26].

Nowadays, both open source and commercial obfuscator tools are mostly targeted for Java [108, 107]. For that platform, the typically used method is to apply the kind of transformations to the binary bytecode representation of the program that do not alter its behaviour, but make the recovery of human readable source code harder for automatic deobfuscators. Another, although questionable, use of obfuscation typically applied to C and C++ sources is the circumvention of open source licenses like GPL. In such cases, the goal of the author is to keep the code in conformance with the license, i.e., to make the source publicly available, but still make the comprehension almost impossible for an outsider.

Although several large software systems are still written in C++, e.g., Symbian, the market-leading smartphone OS and most applications written for it, to date only a few tools have been designed specifically for their protection, and they mostly use simple source code transformations [5, 93]. Since the importance of protecting C++ programs is not negligible, we set ourselves the goal of developing non-trivial obfuscation techniques for C++. Here, we discuss the adaptation of a control flow transformation technique called control flow flattening to the C++ language. Although the general idea has been defined informally in [101], no paper has been published on the adaptation of the technique to a given programming language.

Moreover, to our best knowledge, currently there are no obfuscation techniques which would give protection to programs written in and built from C++ against comprehending the binary code. However, lots of attacks are directed against programs released in binary form to work around or to deactivate their protection. Activation code validation routines and digital rights management (a.k.a. DRM) modules are especially attractive for crackers, to name but a few of their potential targets.

To achieve a kind of protection for C++ originated binaries that is similar to what is already available for Java, two alternatives exist. The first one is to transform the binary code, while the second one is to transform the source in such a way that the transformation has an effect on the compiled binary code as well. Unfortunately, the first alternative has serious drawbacks. First, the transformation of binary programs is a very complex problem which is hard to perform in a safe manner, and second, it has to be adapted to every targeted platform. Thus, in this thesis, we will focus on the second alternative.

In the next chapter, we will discuss the identified problems of adapting control flow flattening to C++ and we give solutions to them. In addition, we will present the complete formal algorithm of the technique. Finally, we will evaluate control flow flattening using a prototype implementation, and we will demonstrate the effect of the algorithm on test programs. Most importantly, in our experiments, we analysed the effect of the algorithm not just at the source code level, but at the binary level as well, thus were able to learn whether a static source-to-source transformation could render the comprehension of the binary code more difficult as well.

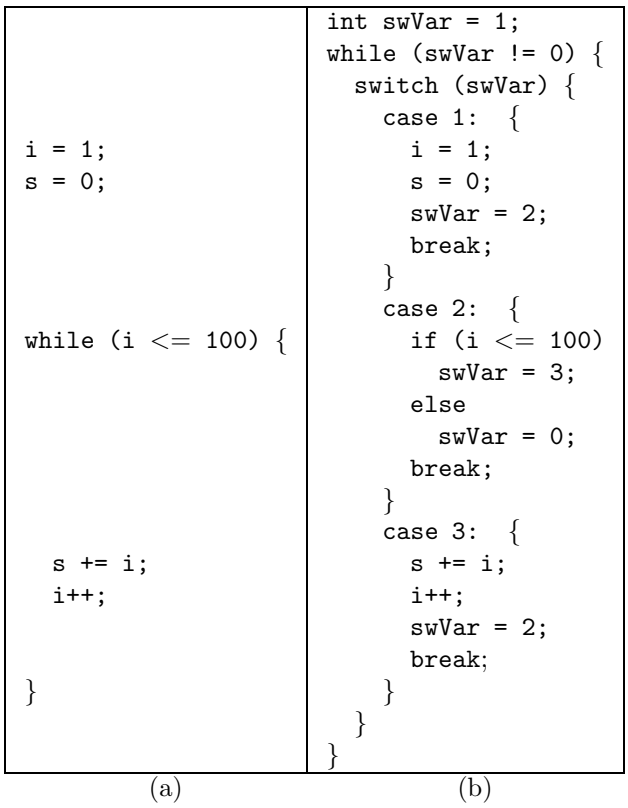
Chapter 13

Control Flow Flattening

13.1 Flattening of C++ Programs

In the case of most real life programs, branches and their targets are easily identifiable due to high level programming language constructs and coding guidelines. In such cases, the complexity of determining the control flow of a function is linear with respect to the number of its basic blocks [84]. The idea behind control flow flattening is to transform the structure of the source code in such a way that the targets of branches cannot be easily determined by static analysis, thus hindering the comprehension of the program.

The basic method for flattening a function is the following. First, we break up the body of the function into basic blocks, and then we put all these blocks, which were originally at different nesting levels, next to each other. These new basic blocks are encapsulated in a selective structure (a `switch` statement in the C++ language) with each block in a separate case, and the selection is in turn encapsulated in a loop. Finally, the correct flow of control is ensured by a control variable representing the state of the program, which is set at the end of each basic block and is used in the predicates of the enclosing loop and selection. An example of this method is given in Figures 13.1 and 13.2. The control flow graphs of the original and the obfuscated code show the changes in the structure of the program, i.e., all the original blocks are at the same level, thus concealing the loop



```
i = 1;
s = 0;

while (i <= 100) {

  s += i;
  i++;

}
```

(a)

```
int swVar = 1;
while (swVar != 0) {
  switch (swVar) {
    case 1:  {
      i = 1;
      s = 0;
      swVar = 2;
      break;
    }
    case 2:  {
      if (i <= 100)
        swVar = 3;
      else
        swVar = 0;
      break;
    }
    case 3:  {
      s += i;
      i++;
      swVar = 2;
      break;
    }
  }
}
```

(b)

Figure 13.1: The effect of control flow flattening on the source code (a: original, b: flattened).

structure of the original program.

According to the above description, the task of flattening a function seems to be quite simple. However, when it comes to applying the idea to a real programming language, we run into certain problems. Below we will discuss the difficulties we encountered during the adaptation of control flow flattening to the C++ language.

As the example in Figure 13.1 makes clear, breaking up loops into basic blocks is not the same as simply splitting the head of the loop from its body. Retaining the same language construct, i.e., `while`, `do` or `for`, in the flattened code would lead to incorrect results, since a single loop head with its body detached definitely cannot reproduce the original behaviour. Thus, for loops, the head of these structures has to be replaced with an `if` statement where

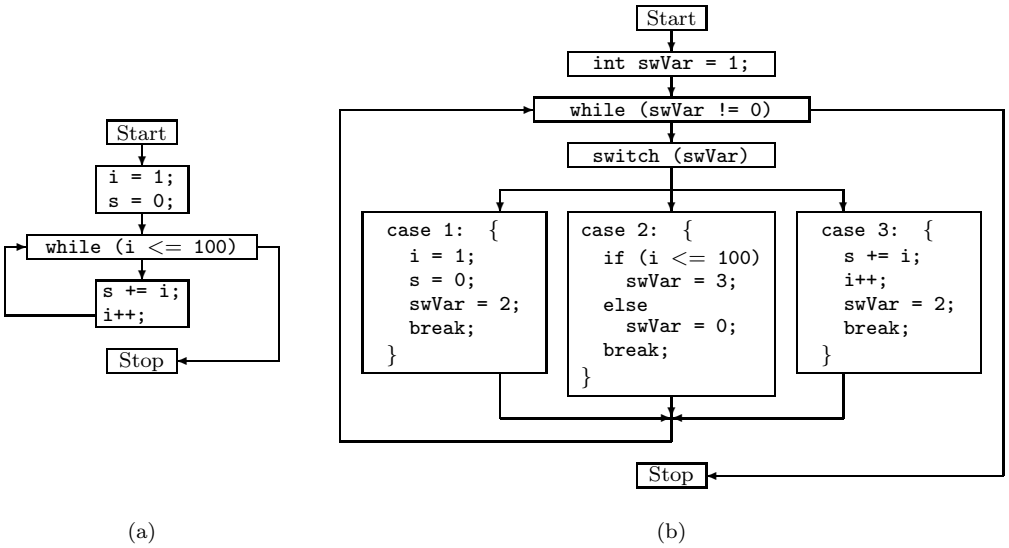
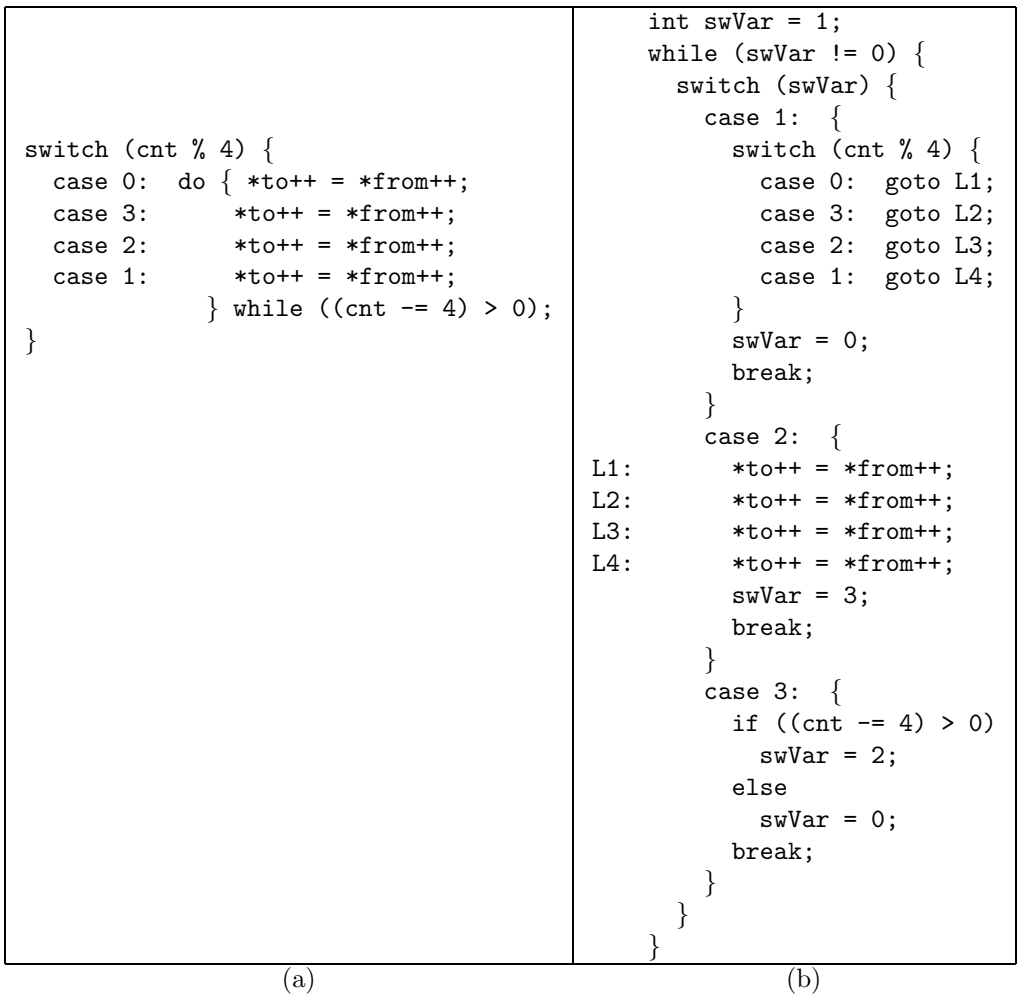


Figure 13.2: The effect of control flow flattening on the control flow graph (a: original, b: flattened).

the predicate is retained from the original contruct and the branches ensure the correct flow of control by assigning appropriate values to the control variable.

Another compound statement that is not easy to deal with is the `switch` construct. In this case, the cause of the problem is the relaxed specification of the `switch` statement, which only requires that the controlled statement of the `switch` be a syntactically valid (compound) statement where case labels can appear as the prefixes of any sub-statements. An interesting example which exploits this lazy specification is Duff's device [97] where loop unrolling is implemented by interlacing the structures of a `switch` and a loop. A slightly modified version of the device and its possible flattened version are given in Figure 13.3.

When it comes to loops and `switch` statements, we must not forget to mention unstructured control transfers either. If left unchanged in the flattened code, `break` and `continue` statements could cause problems, since instead of terminating or restarting the loop or `switch` as they were intended to do, they would terminate or restart the control structure of the flattened code. To avoid this, these kind of instructions have to be replaced in the



```
switch (cnt % 4) {
  case 0:  do { *to++ = *from++;
  case 3:       *to++ = *from++;
  case 2:       *to++ = *from++;
  case 1:       *to++ = *from++;
              } while ((cnt -= 4) > 0);
}
```

(a)

```
    int swVar = 1;
    while (swVar != 0) {
      switch (swVar) {
        case 1:  {
          switch (cnt % 4) {
            case 0:  goto L1;
            case 3:  goto L2;
            case 2:  goto L3;
            case 1:  goto L4;
          }
          swVar = 0;
          break;
        }
        case 2:  {
L1:       *to++ = *from++;
L2:       *to++ = *from++;
L3:       *to++ = *from++;
L4:       *to++ = *from++;
          swVar = 3;
          break;
        }
        case 3:  {
          if ((cnt -= 4) > 0)
            swVar = 2;
          else
            swVar = 0;
          break;
        }
      }
    }
```

(b)

Figure 13.3: Duff's device (a: original code, b: flattened version).

flattened program by assignments to the control variable in such a way that the correct order of execution is ensured. Figure 13.4 gives an example of this replacement.

Compared to C, C++ has an additional control structure, the `try-catch` construct for exception handling. By simply applying the basic idea of control flow flattening to a `try` block, i.e., determining the basic blocks and placing them in the cases of the controlling `switch`, this would violate the logic of exception handling. In such a case, the instructions that would be moved out of the body of the `try` would not be protected anymore by the exception

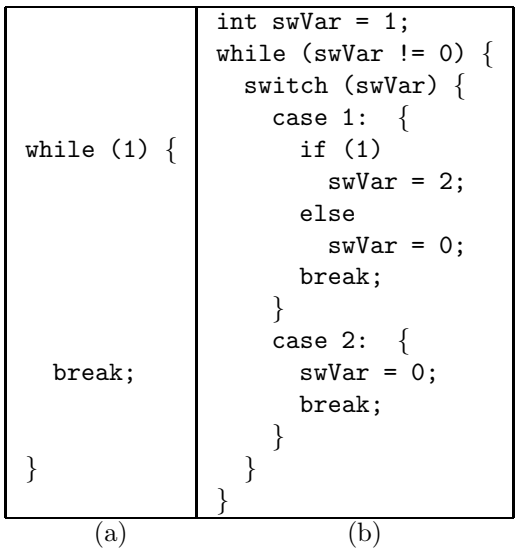
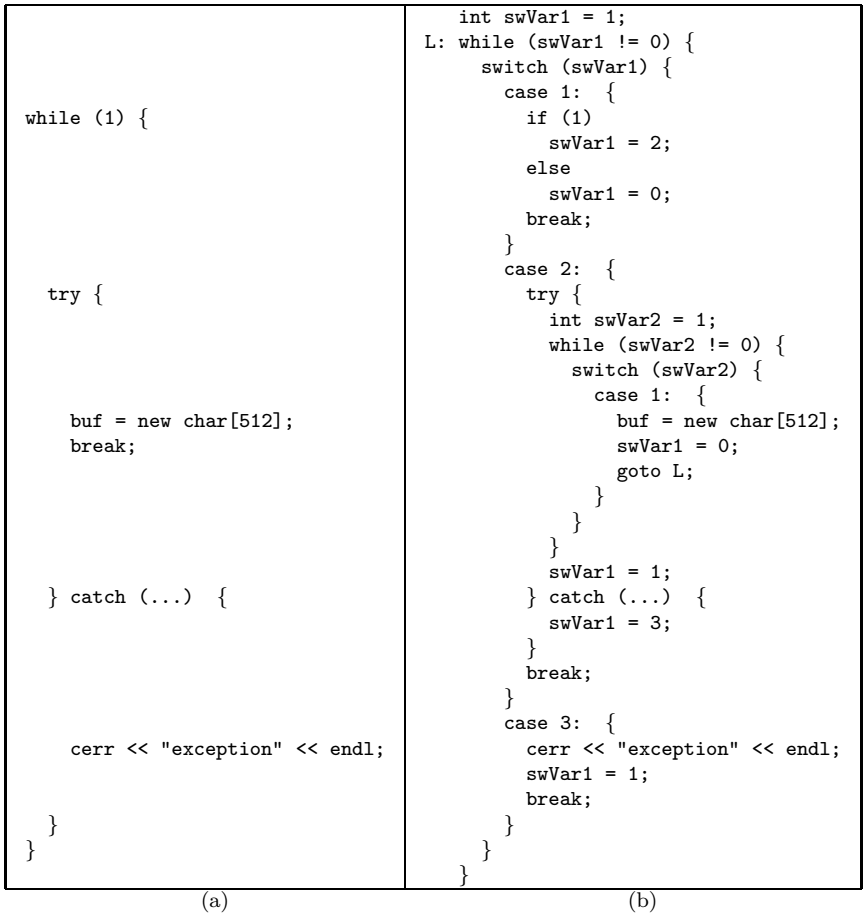


Figure 13.4: Transformation of a loop with unstructured control transfer (a: original code, b: flattened code).

handling mechanism, and thrown exceptions could not be caught by the originally intended handlers. To keep the original behaviour of the program in the flattened version, `try` blocks have to be *flattened independently* of the other parts of the program, which will result in a new `while-switch` control structure which remains under the control of the `try` construct. Thus, the flattening of `try` constructs produces multiple levels of flattened blocks. This causes problems again when an unstructured control transfer has to jump across different levels.

Figure 13.5 shows an example of the multiple levels of flattened blocks created by the transformation of a `try` construct as well as a solution for jumping across levels when it is required by a `break` statement. Although using `goto` statements is usually discouraged by coding guidelines, there are cases when their use is justified [38].



```
while (1) {

  try {

    buf = new char[512];
    break;

  } catch (...)  {

    cerr << "exception" << endl;

  }
}
```

(a)

```
   int swVar1 = 1;
L: while (swVar1 != 0) {
     switch (swVar1) {
       case 1:  {
         if (1)
           swVar1 = 2;
         else
           swVar1 = 0;
         break;
       }
       case 2:  {
         try {
           int swVar2 = 1;
           while (swVar2 != 0) {
             switch (swVar2) {
               case 1:  {
                 buf = new char[512];
                 swVar1 = 0;
                 goto L;
               }
             }
           }
           swVar1 = 1;
         } catch (...)  {
           swVar1 = 3;
         }
         break;
       }
       case 3:  {
         cerr << "exception" << endl;
         swVar1 = 1;
         break;
       }
     }
   }
```

(b)

Figure 13.5: Exception handling with unstructured control transfer (a: original code, b: flattened code).

13.2 The Algorithm

In the following, we will propose an algorithm for flattening the control flow of C++ functions, which solves the problems presented in the previous section. The algorithm expects that the abstract syntax tree of the function-to-be-flattened is available and, after a preprocessing phase, it traverses the tree in one pass along which the obfuscated version of the function is generated.

In the formal description of the algorithm (see Figures 13.6, 13.7, and 13.8), the words in **bold** denote the keywords of the used pseudo-language, the formalised parts are typeset in roman font, while the parts which are easier to

```
levels : stack of ⟨variable, label⟩
breaks : stack of ⟨level, entry⟩
continues : stack of ⟨level, entry⟩

procedure control_flow_flattening (block)
begin
  separate variable declarations from the rest
    of block and output them before all other
    statements
  flatten_block(block)
end

procedure flatten_block (block)
begin
  while_label := unique_identifier()
  switch_variable := unique_identifier()
  entry := unique_number()
  exit := unique_number()
  ⇒ "int " ⊕ switch_variable ⊕ " = " ⊕ entry ⊕
    ";"
  ⇒ while_label ⊕ ":"
  ⇒ "while (" ⊕ switch_variable ⊕ " != " ⊕
    exit ⊕ ") {"
  ⇒ "  switch (" ⊕ switch_variable ⊕ ") {"
  push(levels, ⟨switch_variable, while_label⟩)
  transform_block(block, entry, exit)
  pop(levels)
  ⇒ "  }"
  ⇒ "}"
end
```

```
procedure transform_block (block, entry, exit)
begin
  block_parts[] := split block to parts so that
    each part is either a compound statement
    or a sequence of non-compound statements
  for each part in block_parts do
    part_exit := part is the last ? exit :
      unique_number()
    case type of part of
      block: transform_block(part, entry,
        part_exit)
      if: transform_if(part, entry, part_exit)
      switch: transform_switch(part, entry,
        part_exit)
      while: transform_while(part, entry,
        part_exit)
      do: transform_do(part, entry, part_exit)
      for: transform_for(part, entry, part_exit)
      try: transform_try(part, entry, part_exit)
      sequence: transform_sequence(part, entry,
        part_exit)
    endcase
    entry := part_exit
  endfor
end
```

Figure 13.6: The algorithm of control flow flattening, part one.

explain in free text are in *italics*. The output of the algorithm is a C++ code for which `typewriter` font and double quotes are used. Throughout the algorithm, two symbols are used as well: $\oplus$ denotes string concatenation, while $\Rightarrow$ outputs the result of the algorithm, e.g., to the console or to a file.

The algorithm starts at the *control_flow_flattening* procedure (see Figure 13.6), which first performs a preprocessing on the function. In this step, all the variable declarations that are not at the beginning of the function (i.e., the ones that are preceeded by other statements) are eliminated to avoid visibility problems that would result from the change in the scope of such declarations. So, the declaration of these variables is moved to the beginning of the function, and only their initialisation is left in place, i.e., converted to an assignment. The possible name collisions are resolved by variable renaming.

The actual flattening starts at the procedure *flatten_block* where the construct controlling the control flow is generated. As Figure 13.5 showed in the

```
procedure transform_if (if_stmt, entry, exit)
begin
  switch_variable := top(levels).variable
  then_entry := unique_number()
  else_entry := if_stmt has an else branch ?
    unique_number() : exit
  ⇒ "case " ⊕ entry ⊕ ":  {"
  for each label in labels of if_stmt do
    ⇒ label ⊕ ":"
  endfor
  ⇒ "  if (" ⊕ predicate of if_stmt ⊕ ")"
  ⇒ "    " ⊕ switch_variable ⊕ " = " ⊕
    then_entry ⊕ ";"
  ⇒ "  else"
  ⇒ "    " ⊕ switch_variable ⊕ " = " ⊕
    else_entry ⊕ ";"
  ⇒ "  break;"
  ⇒ "}"
  transform_block(true branch of if_stmt,
    then_entry, exit)
  if if_stmt has an else branch then
    transform_block(else branch of if_stmt,
      else_entry, exit)
  endif
end

procedure transform_switch (switch_stmt, entry,
  exit)
begin
  switch_variable := top(levels).variable
  ⇒ "case " ⊕ entry ⊕ ":  {"
  for each label in labels of switch_stmt do
    ⇒ label ⊕ ":"
  endfor
  ⇒ "  switch (" ⊕ predicate of switch_stmt ⊕
    ") {"
  for each case_label in cases of switch_stmt do
    goto_label := unique_identifier()
    ⇒ "  " ⊕ case_label ⊕ ":"
    ⇒ "    goto " ⊕ goto_label ⊕ ";"
    add a label named goto_label to the
      statement referenced by case_label
  endfor
  ⇒ "  }"
  ⇒ "  " ⊕ switch_variable ⊕ " = " ⊕ exit ⊕ ";"
  ⇒ "  break;"
  ⇒ "}"
  push(breaks, ⟨size(levels), exit⟩)
  transform_block(body of switch_stmt,
    unique_number(), exit)
  pop(breaks)
end
```

```
procedure transform_while (while_stmt, entry,
  exit)
begin
  switch_variable := top(levels).variable
  body_entry := unique_number()
  ⇒ "case " ⊕ entry ⊕ ":  {"
  for each label in labels of while_stmt do
    ⇒ label ⊕ ":"
  endfor
  ⇒ "  if (" ⊕ predicate of while_stmt ⊕ ")"
  ⇒ "    " ⊕ switch_variable ⊕ " = " ⊕
    body_entry ⊕ ";"
  ⇒ "  else"
  ⇒ "    " ⊕ switch_variable ⊕ " = " ⊕ exit ⊕ ";"
  ⇒ "  break;"
  ⇒ "}"
  push(breaks, ⟨size(levels), exit⟩)
  push(continues, ⟨size(levels), entry⟩)
  transform_block(body of while_stmt,
    body_entry, entry)
  pop(breaks)
  pop(continues)
end

procedure transform_do (do_stmt, entry, exit)
begin
  switch_variable := top(levels).variable
  test_entry := unique_number()
  body_entry := unique_number()
  ⇒ "case " ⊕ test_entry ⊕ ":  {"
  ⇒ "  if (" ⊕ predicate of do_stmt ⊕ ")"
  ⇒ "    " ⊕ switch_variable ⊕ " = " ⊕
    body_entry ⊕ ";"
  ⇒ "  else"
  ⇒ "    " ⊕ switch_variable ⊕ " = " ⊕ exit ⊕ ";"
  ⇒ "  break;"
  ⇒ "}"
  ⇒ "case " ⊕ entry ⊕ ":  {"
  for each label in labels of do_stmt do
    ⇒ label ⊕ ":"
  endfor
  ⇒ "  " ⊕ switch_variable ⊕ " = " ⊕
    body_entry ⊕ ";"
  ⇒ "  break;"
  ⇒ "}"
  push(breaks, ⟨size(levels), exit⟩)
  push(continues, ⟨size(levels), test_entry⟩)
  transform_block(body of do_stmt, body_entry,
    test_entry)
  pop(breaks)
  pop(continues)
end
```

Figure 13.7: The algorithm of control flow flattening, part two.

previous section, sometimes it is necessary to jump across different levels of flattened blocks. To aid this, the controlling loop is annotated with a label, and this label together with the name of the control variable is pushed to a

```
procedure transform_for (for_stmt, entry, exit)
begin
  switch_variable := top(levels).variable
  test_entry := unique_number()
  inc_entry := unique_number()
  body_entry := unique_number()
  ⇒ "case " ⊕ entry ⊕ ":  {"
  for each label in labels of for_stmt do
    ⇒ label ⊕ ":"
  endfor
  ⇒ "  " ⊕ initialization part of for_stmt
  ⇒ "  " ⊕ switch_variable ⊕ " = " ⊕ test_entry
    ⊕ ";"
  ⇒ "  break;"
  ⇒ "}"
  ⇒ "case " ⊕ test_entry ⊕ ":  {"
  ⇒ "  if (" ⊕ predicate of for_stmt ⊕ ")"
  ⇒ "    " ⊕ switch_variable ⊕ " = " ⊕
    body_entry ⊕ ";"
  ⇒ "  else"
  ⇒ "    " ⊕ switch_variable ⊕ " = " ⊕ exit ⊕ ";"
  ⇒ "  break;"
  ⇒ "}"
  ⇒ "case " ⊕ inc_entry ⊕ ":  {"
  ⇒ "  " ⊕ increment part of for_stmt
  ⇒ "  " ⊕ switch_variable ⊕ " = " ⊕ test_entry
    ⊕ ";"
  ⇒ "  break;"
  ⇒ "}"
  push(breaks, ⟨size(levels), exit⟩)
  push(continues, ⟨size(levels), inc_entry⟩)
  transform_block(body of for_stmt, body_entry,
    inc_entry)
  pop(breaks)
  pop(continues)
end

procedure transform_try (try_stmt, entry, exit)
begin
  switch_variable := top(levels).variable
  ⇒ "case " ⊕ entry ⊕ ":  {"
  for each label in labels of try_stmt do
    ⇒ label ⊕ ":"
  endfor
  ⇒ "  try {"
  flatten_block(body of try_stmt)
  ⇒ "  }"
  for each handler in catch handlers of
    try_stmt do
    ⇒ "  catch (" ⊕ parameter of handler ⊕ ") {"
    flatten_block(body of handler)
    ⇒ "  }"
  endfor
  ⇒ "  " ⊕ switch_variable ⊕ " = " ⊕ exit ⊕ ";"
  ⇒ "  break;"
  ⇒ "}"
end
```

```
procedure transform_sequence (sequence, entry,
  exit)
begin
  ⇒ "case " ⊕ entry ⊕ ":  {"
  for each stmt in sequence do
    for each label in labels of stmt do
      ⇒ label ⊕ ":"
    endfor
    case type of stmt of
      continue:
        ⇒ levels[top(continues).level].variable ⊕
          " = " ⊕ top(continues).entry ⊕ ";"
        if top(continues).level <> size(levels) then
          ⇒ "goto " ⊕
            levels[top(continues).level].label ⊕ ";"
        else
          ⇒ "break;"
        endif
      break:
        ⇒ levels[top(breaks).level].variable ⊕
          " = " ⊕ top(breaks).entry ⊕ ";"
        if top(breaks).level <> size(levels) then
          ⇒ "goto " ⊕
            levels[top(breaks).level].label ⊕ ";"
        else
          ⇒ "break;"
        endif
      otherwise:
        ⇒ stmt
    endcase
  endfor
  ⇒ top(levels).variable ⊕ " = " ⊕ exit ⊕ ";"
  ⇒ "break;"
  ⇒ "}"
end
```

Figure 13.8: The algorithm of control flow flattening, part three.

stack (*levels*) every time a new level is created.

The procedure *transform_block*, called from the *flatten_block*, is responsible for breaking up a block into compound statements and sequences of non-compound statements, while the other *transform* procedures do the obfuscation of these block parts according to their type. The procedure *transform_if* in Figure 13.7 is a good example of how compound statements are obfuscated: a new case is generated in the controlling `switch` for the head of the selection, while the branches are handled by calling *transform_block* on them recursively. The procedure *transform_while* works in a similar way, except that before recursively calling *transform_block*, the case labels where the execution shall continue on a `break` or `continue` statement are pushed to two stacks, *breaks* and *continues*, respectively. Along with the case labels, the depth of the actual level of flattening, i.e., the number of entries in the *levels* stack, is pushed to both stacks as well. The same approach is used to transform `do` and `for` statements. The procedure *transform_switch* also uses stacking to deal with unstructured control transfer, but just the *breaks* stack is used, since `continue` statements have no effect on a `switch`.

The last type of compound statements to be transformed is `try`. As mentioned in the previous section, this construct requires the use of multiple levels of flattened blocks. Thus, contrary to the previous procedures, *transform_try* in Figure 13.8 calls *flatten_block* recursively instead of *transform_block*.

Finally, the procedure *transform_sequence* is the one that handles simple statements, and this is where the stacks managed in *flatten_block* (*levels*) and in some of the *transform* procedures (*breaks*, *continues*) are utilised. All `break` and `continue` statements are rewritten to an assignment to the control variable – more precisely, to the appropriate control variable. The *levels* stack together with either the *breaks* or the *continues* stack determine which variable is to be used. In addition, if the stacks indicate that the control has to cross levels of flattening, a `goto` instruction is inserted, as can be seen in the example in Figure 13.5.

13.3 Experimental Results

To evaluate how effective control flow flattening is in protecting either the source code or the binary program compiled from the obfuscated source, we used the following scenario. First, we collected a benchmark which consisted of 23 functions selected from programs of the Java-is-faster-than-C++ Benchmark [75], the C version of the LINPACK Benchmark [85] and LDA-C [20]. These functions were then obfuscated using a prototype tool which implements the obfuscation technique introduced in the previous section on the basis of the CAN C++ analyzer of the Columbus framework [39]. Before and after obfuscation, we computed McCabe's cyclomatic complexity metric [81] from the source representation of each function to measure the change in their complexity and comprehensibility. Once the obfuscated versions of the sources were generated and the complexity metrics were computed, we compiled both the original and the obfuscated codes using GCC 4.0.2 for the ARM target. The resulting binaries were analysed using the same tool that we applied for slicing binary executables (as described in Part II) and we built control flow graphs from the binary representations of the benchmark functions. These CFGs were then used to compute McCabe's metric for the original and obfuscated codes. This data was in turn used to measure the change in the complexity and comprehensibility of the binary programs.

As we mentioned above, first, we collected a benchmark. Table 13.1 lists the names of the selected functions and gives their size as the number of their effective lines of code (ELOC). As the table shows, our benchmark contains functions of only 1 line (ack in ackermann.cpp) up to functions consisting of 78 effective lines (dgefa in linpack.cpp).

The effect of obfuscation on complexity (and thus, on comprehensibility) is presented in Table 13.2. Here, we investigated how McCabe's metric changes with source code and in the case of two binary versions which were compiled from source using two different optimisation settings, -O0 and -O2. As the second column of the table shows, we obtained a significant, 4.63-fold increase in the complexity of the source code on average. Moreover, the data in the third and fourth columns supported our assumption, i.e., that the

Table 13.1: The benchmark used to evaluate the effects of control flow flattening.

Function	**ELOC**
dgefa (linpack.cpp)	78
read_data (lda-data.c)	34
main (moments.cpp)	26
main (sumcol.cpp)	26
main (almabench.cpp)	25
main (wc.cpp)	24
lda_mle (lda-model.c)	24
save_lda_model (lda-model.c)	23
main (sieve.cpp)	20
main (nestedloop.cpp)	18
matgen (linpack.cpp)	18
new_lda_model (lda-model.c)	17
main (matrix.cpp)	16
argmax (utils.c)	15
deep (penta.cpp)	13
mmult (matrix.cpp)	13
main (random.cpp)	13
log_sum (utils.c)	12
digamma (utils.c)	10
anpm (almabench.cpp)	7
radecdist (almabench.cpp)	7
gen_random (random.cpp)	5
ack (ackermann.cpp)	1

complexity of the binary programs increases as a result of the obfuscation of the source code. The increase in McCabe's metric measured on binary programs is similar to the increase measured on the sources, i.e., 5.19-fold and 3.26-fold, on average, for -O0 and -O2-optimised binaries, respectively. The somewhat smaller increase in the case of -O2-optimised binaries can be attributed to the strong optimisation techniques applied by GCC. However, a more than threefold increase can still be considered significant, and it shows that compiler optimisations do not eliminate the effects of the source obfuscation technique.

Statistical analysis also supports our hypothesis that the increase in McCabe's metric on the binaries can be attributed to the obfuscation of the source code. As Table 13.3 shows, the increase in the complexity metric

Table 13.2: The effect of control flow flattening on McCabe's complexity metric.

Function	Source	Binary -O0	Binary -O2
dgefa (linpack.cpp)	16→65 (4.06×)	16→65 (4.06×)	23→58 (2.52×)
read_data (lda-data.c)	4→17 (4.25×)	4→16 (4.00×)	5→14 (2.80×)
main (moments.cpp)	5→28 (5.60×)	4→33 (8.25×)	18→46 (2.56×)
main (sumcol.cpp)	3→32 (10.67×)	2→41 (20.50×)	4→46 (11.50×)
main (almabench.cpp)	4→25 (6.25×)	4→28 (7.00×)	8→37 (4.63×)
main (wc.cpp)	9→20 (2.22×)	8→31 (3.88×)	14→40 (2.86×)
lda_mle (lda-model.c)	5→19 (3.80×)	4→18 (4.50×)	8→15 (1.88×)
save_lda_model (lda-model.c)	3→15 (5.00×)	3→14 (4.67×)	6→11 (1.83×)
main (sieve.cpp)	8→31 (3.88×)	8→33 (4.13×)	13→39 (3.00×)
main (nestedloop.cpp)	9→39 (4.33×)	9→42 (4.67×)	9→47 (5.22×)
matgen (linpack.cpp)	7→28 (4.00×)	7→28 (4.00×)	6→23 (3.83×)
new_lda_model (lda-model.c)	3→15 (5.00×)	3→14 (4.67×)	5→12 (2.40×)
main (matrix.cpp)	3→16 (5.33×)	3→19 (6.33×)	8→30 (3.75×)
argmax (utils.c)	3→12 (4.00×)	3→11 (3.67×)	3→ 9 (3.00×)
deep (penta.cpp)	5→20 (4.00×)	5→19 (3.80×)	7→16 (2.29×)
mmult (matrix.cpp)	4→20 (5.00×)	4→19 (4.75×)	4→16 (4.00×)
main (random.cpp)	3→14 (4.67×)	3→17 (5.67×)	8→28 (3.50×)
log_sum (utils.c)	2→ 9 (4.50×)	2→ 8 (4.00×)	2→ 7 (3.50×)
digamma (utils.c)	1→ 4 (4.00×)	1→ 3 (3.00×)	1→ 1 (1.00×)
anpm (almabench.cpp)	3→ 9 (3.00×)	3→ 8 (2.67×)	2→ 6 (3.00×)
radecdist (almabench.cpp)	2→ 8 (4.00×)	2→ 7 (3.50×)	2→ 6 (3.00×)
gen_random (random.cpp)	1→ 7 (7.00×)	1→ 6 (6.00×)	1→ 2 (2.00×)
ack (ackermann.cpp)	3→ 6 (2.00×)	3→ 5 (1.67×)	13→13 (1.00×)

on source code closely correlates with the increase on both binary versions. Our analysis of the raw data also confirms that the effect of the algorithm on complexity is linearly proportional to the original complexity. For source code and for both binary versions, Figure 13.9 shows how the complexity of the obfuscated code varies as a function of the complexity of the original code and also the lines fitted via linear regression on the data.

In addition to the effect on complexity, we measured the effect of control flow flattening on resource consumption as well. Therefore, we examined the change in the size of the benchmark functions. Since making a comparison of the lines of code metric for a hand-written and an automatically generated code is not a fair one, we decided to count the nodes in the abstract syntax tree (AST) representation of the functions to measure the effect of flattening

Table 13.3: Pearson correlation of the increases in McCabe's metric in the cases of source code, -O0-optimised binary code, and -O2-optimised binary code. All correlations are significant at the 0.01 level (2-tailed).

	Source	**Binary -O0**	**Binary -O2**
Source	1	0.905	0.747
Binary -O0	0.905	1	0.872
Binary -O2	0.747	0.872	1

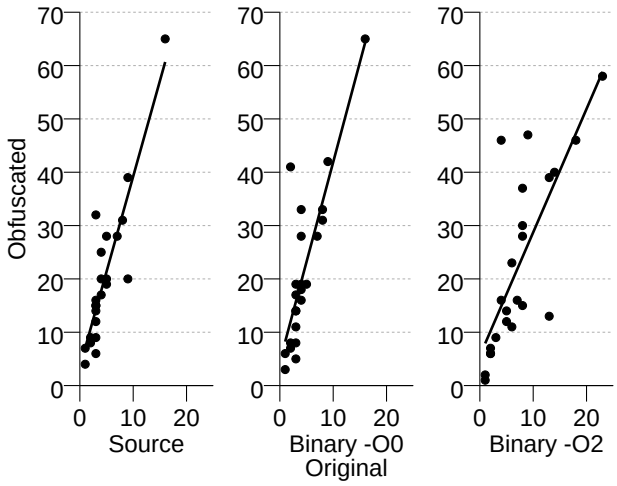


Figure 13.9: The relationship between the complexities of the original and the flattened code.

on the source size. The change in the binary code size is easier to measure, and the symbols in the ARM binary executable files help to determine the size of the functions in bytes. The results listed in Table 13.4 show that the size of the obfuscated sources is about twice as big as the original size on average, while the size increase of the -O0 and -O2-optimised binary code is only 1.55-fold and 1.57-fold on average.

In addition to the analysis of the static code size, we examined the effect of control flow flattening on a dynamic attribute as well: we counted the number of executed instructions in some of the benchmark functions using an enhanced version of the ARM simulator of GDB. In Table 13.5, we

Table 13.4: The effect of control flow flattening on program size.

Function	**Source (AST)**	**Binary -O0 (bytes)**	**Binary -O2 (bytes)**
dgefa (linpack.cpp)	494→894 (1.81×)	2352→3572 (1.52×)	1260→2604 (2.07×)
read_data (lda-data.c)	198→286 (1.44×)	668→ 916 (1.37×)	388→ 560 (1.44×)
main (moments.cpp)	105→355 (3.38×)	1252→2080 (1.66×)	2512→3076 (1.22×)
main (sumcol.cpp)	94→392 (4.17×)	1386→2072 (1.49×)	2380→2840 (1.19×)
main (almabench.cpp)	90→257 (2.86×)	772→1360 (1.76×)	1332→1744 (1.31×)
main (wc.cpp)	99→273 (2.76×)	868→1436 (1.65×)	1484→1892 (1.27×)
lda_mle (lda-model.c)	101→196 (1.94×)	528→ 796 (1.51×)	336→ 484 (1.44×)
save_lda_model (lda-model.c)	103→182 (1.77×)	404→ 620 (1.53×)	316→ 444 (1.41×)
main (sieve.cpp)	93→287 (3.09×)	1132→1760 (1.55×)	1900→2460 (1.29×)
main (nestedloop.cpp)	89→320 (3.60×)	788→1548 (1.96×)	1084→1900 (1.75×)
matgen (linpack.cpp)	126→267 (2.12×)	764→1120 (1.47×)	360→ 684 (1.90×)
new_lda_model (lda-model.c)	77→151 (1.96×)	304→ 524 (1.72×)	168→ 340 (2.02×)
main (matrix.cpp)	112→228 (2.04×)	848→1288 (1.52×)	1524→1816 (1.19×)
argmax (utils.c)	34→ 92 (2.71×)	204→ 380 (1.86×)	96→ 204 (2.13×)
deep (penta.cpp)	79→178 (2.25×)	572→ 848 (1.48×)	220→ 508 (2.31×)
mmult (matrix.cpp)	61→163 (2.67×)	316→ 608 (1.92×)	128→ 368 (2.88×)
main (random.cpp)	56→155 (2.77×)	648→1056 (1.63×)	1248→1504 (1.21×)
log_sum (utils.c)	39→ 78 (2.00×)	368→ 504 (1.37×)	168→ 288 (1.71×)
digamma (utils.c)	81→ 93 (1.15×)	1116→1172 (1.05×)	752→ 752 (1.00×)
anpm (almabench.cpp)	27→ 61 (2.26×)	288→ 412 (1.43×)	156→ 256 (1.64×)
radecdist (almabench.cpp)	92→128 (1.39×)	596→ 712 (1.19×)	304→ 392 (1.29×)
gen_random (random.cpp)	18→ 54 (3.00×)	276→ 452 (1.64×)	140→ 216 (1.54×)
ack (ackermann.cpp)	24→ 35 (1.46×)	172→ 224 (1.30×)	408→ 408 (1.00×)

present the effect of obfuscation on the runtime behaviour for a subset of all the benchmark functions. (Unfortunately, although all benchmark programs could be compiled for the ARM target, because of porting problems, not all of them executed correctly on the simulator.) As the table shows, the increase in the number of executed instructions is 2.03-fold and 2.39-fold on average for the -O0 and -O2-optimised programs.

We should remark here that in a real situation flattening is not expected to be performed on the whole program but only on some selected critical functions or modules, as mentioned earlier. This means that in real applications both the static and the dynamic effects on the resource consumption of the whole program should be much smaller.

Table 13.5: The effect of control flow flattening on the number of executed instructions.

Function	**Binary -O0**	**Binary -O2**
read_data (lda-data.c)	79278→ 135992 (1.72×)	35422→ 69388 (1.96×)
main (wc.cpp)	8968073→19948317 (2.22×)	5432144→12235649 (2.25×)
lda_mle (lda-model.c)	15969282→28325635 (1.77×)	9077355→14728182 (1.62×)
save_lda_model (lda-model.c)	4781282→11018150 (2.30×)	2286764→ 5613102 (2.45×)
main (sieve.cpp)	26461157→80562761 (3.04×)	8358569→57199282 (6.84×)
new_lda_model (lda-model.c)	3949918→10186780 (2.58×)	1247460→ 4989484 (4.00×)
main (matrix.cpp)	670→ 1664 (2.48×)	701→ 1205 (1.72×)
argmax (utils.c)	166492→ 398204 (2.39×)	89154→ 209508 (2.35×)
mmult (matrix.cpp)	25852143→51781473 (2.00×)	5840373→19763133 (3.38×)
main (random.cpp)	510112→ 1110196 (2.18×)	390185→ 660233 (1.69×)
log_sum (utils.c)	1415248→ 2499893 (1.77×)	724684→ 1339416 (1.85×)
digamma (utils.c)	7702020→ 8044332 (1.04×)	5134680→ 5134680 (1.00×)
gen_random (random.cpp)	2010000→ 3120020 (1.55×)	990000→ 1320002 (1.33×)
ack (ackermann.cpp)	18392568→26026172 (1.42×)	3590986→ 3590986 (1.00×)

Chapter 14

Conclusions

We realised that there was a definite need for the obfuscation of C++ programs and thus, in this part of the thesis, we described the adaptation of a technique called control flow flattening. We identified the problems that occurred during the adaptation and proposed solutions for them. Moreover, we also gave a formal description of an algorithm that performed control flow flattening based on these solutions. The algorithm tells us how to transform general control structures and how to deal with unstructured control transfers. In addition, the technique flattens exception handling contructs as well. Since the transformed control structures are quite similar in other widespread languages, the algorithm can be used as a starting point when control flow flattening has to be adapted.

Furthermore, we found that an important motivation for program code obfuscation was to hinder the reverse engineering of the binary program to a human readable source form. At least, this is true for Java. However, we also found that no such solution exists for C++, although there is a need for this kind of protection. Thus, we presented the idea of applying source-to-source transformations in order to cause a detrimental effect on the comprehensibility of the compiled binaries.

In the previous chapter, we performed experiments with a working prototype of control flow flattening. In the experiments, we measured the effect of flattening on (both source and binary) code complexity and on resource

consumption as well. Our results confirm that control flow flattening causes a significant increase in the complexity of both the source and the binary code, even in the presence of optimisations. These results indicate that the transformation will make the code difficult to understand, while at the same time, the increase in the resource consumption remains acceptable. Moreover, they show that not just the source code but even binary programs can be obfuscated in a platform independent manner using appropriate source code transformation techniques.

Part IV

Appendices

Chapter 15

Summary

15.1 Summary in English

In this thesis, we presented three areas of the domain of program code analysis and manipulation, namely the theoretical foundations of program slicing, the application of program slicing to binary programs, and the obfuscation of programs written in C++ language. The main body of the thesis is split up into three parts according to the above three topics. Below, we give a summary of each part.

In Part I, we presented results concerning the theory of program slicing. Program projection theory was used to uncover the precise relationship between various forms of dynamic slicing and static slicing. It had previously been thought that there were only two nodes in the subsumption relationship between static and dynamic slicing. That is, it was thought that the dynamic slicing criterion merely adds the input sequence to the static criterion and this is all that there is to the difference between the two.

However, the results of the study presented here demonstrate that the original dynamic slicing criterion introduced by Korel and Laski contains two additional aspects over and above the input sequence. The discovery of these two additional criteria led to the creation of a unified framework of program projection theory. With the help of this unified framework, these criteria are shown to be orthogonal components of the original dynamic slicing definition.

Based on our unified framework, two forms of subsumption relationship were considered. The first is the relationship between the semantic properties of a slice, as captured in the equivalence maintained by slicing. The second relationship concerns the relationship between slicing techniques. Thus, the first subsumption relationship is simply about the semantic projections denoted by the different forms of the slicing criteria, while the second concerns the slices which may be produced when this semantic requirement is combined with a syntactic ordering. The results show that the two lattices so-constructed are isomorphic.

In addition, the syntactic ordering relationship between slicing techniques for static and dynamic slicing was also investigated. It was shown that syntactic ordering is a mirror image of the subsumes relationship, leading to an inverted but isomorphic lattice of inter-technique relationships.

In Part II, we described how interprocedural slicing can be applied to binary executables. First, we discussed the problems associated with the control flow analysis of binary programs, and then we presented a conservative dependence graph-based slicing approach along with its improved versions. We experimented with two static improvements, and we also outlined how superfluous edges could be removed from the statically computed call graph with the help of dynamically gathered information.

We evaluated both approaches on programs compiled for ARM architecture with the help of a prototype implementation of the above-described methods. Using the conservative approach, we achieved an interprocedural slice size of 52%-71% on average and a 1%-4% reduction using the static improvements. The experiments with the dynamic approach demonstrated that the slice size could be dramatically further reduced if the analysed application made extensive use of indirect function calls. Even though the resulting improved call graph may be unsafe, it may work well in situations where the safety of slices is not critical, e.g. in some debugging scenarios.

Finally, in Part III, we described the adaptation of a code obfuscation technique called control flow flattening to C++ programs. We identified the problems that occurred during the adaptation and proposed solutions for them. Moreover, we gave a formal description of an algorithm that per-

formed control flow flattening based on these solutions. The algorithm tells us how to transform general control structures and how to deal with unstructured control transfers. In addition, the technique flattens exception handling constructs. Since the transformed control structures are quite similar in other widespread languages, the algorithm can be used as a starting point when control flow flattening has to be adapted.

Furthermore, we found that an important motivation for program code obfuscation is to hinder the reverse engineering of the binary program to a human readable source form. However, we also learned that no such solution exists for C++, although there is a need for this kind of protection. Thus, we presented the idea of applying source-to-source transformations in order to create a detrimental effect on the comprehensibility of the compiled binaries.

The idea of C++ code obfuscation was not only presented theoretically, but we performed experiments with a working prototype of control flow flattening as well. In the experiments, we measured the effect of flattening on (both source and binary) code complexity. Our results show that control flow flattening causes a significant increase in the complexity of both the source and the binary code, even in the presence of optimisations. These results also tell us that the transformation will make the code difficult to understand, while at the same time the increase in the resource consumption remains acceptable. Moreover, this demonstrates that not only the source code but even binary programs can be obfuscated in a platform independent manner using appropriate source code transformation techniques.

15.2 Summary in Hungarian

A disszertációban bemutattunk a programkód elemzésének és módosításának területéről három részterületet. Ezek a következők: a programszeletelés elméleti alapjai, a programszeletelés alkalmazása bináris programokra, valamint C++ nyelvű programok obfuszkálása. Ez a három téma a disszertáció szerkezetében is tükröződik, a dolgozat három fő részből áll. A következőkben összegezzük mindhárom részt.

A dolgozat első részében a programszeletelés elméletével kapcsolatos ered-

mények találhatók. A programprojekció elméletét használtuk fel a különféle dinamikus és statikus szeletelések közötti kapcsolatok pontos feltárására. Korábban az volt az általános vélekedés, hogy a statikus és dinamikus szeletelés között a fölérendeltségi reláció mindössze kételemű. Azaz a legtöbben úgy vélték, hogy a dinamikus szeletelési kritérium csupán a program bemenetével bővíti a statikus kritériumot, és a két kritérium között nincs is más eltérés.

Az itt leírt vizsgálódások azonban rámutatnak, hogy a Korel és Laski által bevezetett eredeti dinamikus szeletelési kritérium a bemeneten túl két további elemet is tartalmaz. A két új kritérium felfedezése vezetett az egységes programprojekció-elméleti keretrendszer létrejöttéhez. Az egységes keretrendszer segítségével megmutattuk, hogy a fenti kritériumok az eredeti dinamikus szeletelési definíciónak ortogonális elemei.

Az egyéges keretrendszer segítségével a fölérendeltségi reláció két formáját is megvizsgáltuk. Ezek közül az első a programszeletek szemantikus jellemzői közötti reláció, amely jellemzőket a szeletelések által megtartott ekvivalenciák írnak le. A második reláció a szeletelési módszerek közötti kapcsolatot ragadja meg. Más szóval az első fölérendeltségi reláció pusztán a különféle szeletelési kritériumok által meghatározott szemantikus projekciókkal foglalkozik, míg a második magukat a szeleteket veszi figyelembe, amelyek előállításához a szemantikus feltételhez egy szintaktikus rendezést is csatolni kell. Az eredményeink azt mutatják, hogy a relációk által meghatározott hálók izomorfak.

A fentieken túl a statikus és dinamikus szeletelési módszerek szintaktikus rendezési relációját is megvizsgáltuk. Ennek eredménye rávilágított arra, hogy a szintaktikus rendezés a fölérendeltségi reláció tükörképe. Így tehát a programszeletelési technikák kettő, egymáshoz képest invertált, ám izomorf hálóját kaptuk.

A disszertáció második részében tárgyaltuk az interprocedurális szeletelés bináris programokra való alkalmazását. Először bemutattuk a bináris programok vezérlési folyamanalízisével járó problémákat, majd leírtunk egy konzervatív, függőségi gráf alapú szeletelési módszert. Emellett a módszernek javításait is bemutattuk. Kísérleteztünk kétféle statikus javítással, majd

megmutattuk, hogy miként lehet dinamikusan gyűjtött információk segítségével a statikusan épített hívási gráfból a felesleges éleket eltávolítani.

Mind a statikus, mind a dinamikus módon javított módszert kipróbáltuk ARM architektúrára fordított programokon. Ehhez a bemutatott módszerek egy prototípus implementációját használtuk fel. A konzervatív megközelítéssel átlagosan 52%-71%-os interprocedurális szeletméretet értünk el, míg a statikus javítások 1%-4% méretcsökkenést eredményeztek. A dinamikus megközelítéssel végzett kísérletek azt mutatták, hogy a szeletek mérete jelentősen tovább csökkenthető, amennyiben az elemzett alkalmazás nagy számban használ indirekt függvényhívásokat. Ugyan az így kapott hívási gráf pontossága nem garantálható, a megközelítés jól használható marad olyan esetekben, amikor a szeletek pontossága nem kritikus, például bizonyos hibakeresési szkenáriók esetén.

Végezetül a harmadik részben bemutattuk a vezérlési folyamlapításnak nevezett kódobfuszkálási technika C++ programokra való adaptálását. Azonosítottuk az adaptálás során felmerült problémákat, majd megoldásokat javasoltunk rájuk. Emellett a javasolt megoldásokon alapulva egy vezérlési folyamlapító algoritmus formális leírását is megadtuk. Ez az algoritmus bemutatja, hogy hogyan kell az általános vezérlési szerkezeteket átalakítani, valamint hogy miként kell kezelni a nem struktúrált vezérlésátadásokat. A módszer mindemellett kivételkezelő szerkezetek lapítására is képes. Mivel a módszer által lekezelt és átalakított vezérlési szerkezetek más, szintén elterjedt nyelvekben is igen hasonlóak, ezért az itt leírt algoritmus jó kiindulási alap lehet olyankor, amennyiben a vezérlési folyamlapítási technikát más nyelvre kell adaptálni.

Kutatásunk során észrevettük, hogy sokszor azért kerül sor a programkód obfuszkálására, hogy megnehezítsék a bináris programok visszafordítását ember számára is érthető forráskód formába. Azt is észrevettük azonban, hogy ilyen megoldás nem létezik C++ nyelvű programokra, habár van igény ilyen jellegű programvédelemre. Ennek okán kísérleteztünk azzal az elgondolással, hogy pusztán forráskód transzformációkkal meg lehet-e nehezíteni a lefordított binárisok megérthetőségét.

A C++ nyelvű programok obfuszkálását nem csupán elméleti síkon tár-

gyaltuk, hanem a vezérlési folyamlapítás egy működő prototípus-implementációjával kísérleteket is végeztünk. A kísérletek során megmértük, hogy a lapítás milyen módon hat a kód bonyolultságára (mind forráskód, mind bináris kód tekintetében). Az eredményeink azt mutatják, hogy a vezérlési folyamlapítás jelentős bonyolultságnövekedést eredményez mind forráskódban, mind binárisban. Az eredmények arra is utalnak, hogy az elvégzett átalakítás megnehezíti a kód megértését, miközben az erőforrásigény növekedése még elfogadható marad. A fentiekből továbbá az is következik, hogy megfelelő forráskód-átalakító technikák használatával nem csak a forráskód, de bináris programok obfuszkálása is kivitelezhető platformfüggetlen módon.

15.3 Main Results of the Thesis and Contributions of the Author

In this thesis, four main results are stated. As the thesis consists of three main parts, the results are also separated into three parts. In the list below, the author's contributions to these results are clearly stated.

The Theory of Slicing

The main results of the first part of the thesis include the creation of a unified framework of program projection theory and the analysis of the relationships between the different forms of slicing that was made possible by the unified framework. (Note that these results use program projection theory and its application to Weiser's static slicing as a background, but these are not the works of the author.) The results, which are based on publications [12, 11, 14, 13], are presented in Chapters 4 and 5 respectively.

1. The Unified Framework of Program Projection Theory

Based on the results of a comparison of Weiser's static slicing and Korel and Laski's dynamic slicing, the author found that the dynamic slicing criterion does not merely add the input sequence to the static criterion but

contains two additional aspects as well. The discovery of these two additional components of the dynamic criterion allowed the author to create a unified equivalence and a unified framework of program projection theory. Thus, the author was able to put the two slicing approaches, i.e., dynamic and static slicing, into one framework. The created framework not only allowed the author to re-define the existing and well-known slicing techniques of Weiser, and Korel and Laski, but it also led to the identification of six new possible forms of slicing that were hitherto unknown in the literature. (Note that the discussion of these new forms of slicing is a joint work of the author and his co-authors.)

2. Analysis of the Relationships between Forms of Slicing

The author defined a subsumption relationship between the semantic aspect of forms of slicing and, using the unified equivalence, he showed that the semantic parts of the eight forms of slicing described in this thesis form a lattice. In addition, the author also showed that when not just the semantic aspect but also the syntactic component of slicing techniques are considered, the subsumption relationship between the eight forms of slicing does not change.

Since the size of slices is of great importance in every slicing application, the author chose to investigate the minimal slices allowed by slicing techniques. The author found that slicing techniques can be ordered based on sets of minimal slices and that the so-resulting ordering is the dual of the subsumption relationship. The author showed that over the eight previously mentioned forms of slicing, this ordering forms a lattice that is the mirror image of the lattice of the subsumption relationship.

Slicing of Binary Programs

The main result of the second part of the thesis is the description of the dependence graph-based slicing of binary executables, which can be found in Chapters 8, 9, and 10. These results were published in papers [64, 63].

3. Dependence Graph-based Slicing of Binary Executables

Similar to other code analysing techniques, dependence graph-based slicing requires a control flow graph. Thus, the author decided to explore the problems of control flow analysis of binary programs and proposed solutions as well. In addition to a discussion of this analysis, the method of building program and system dependence graphs for binaries is given as well. (Note, however, that this method is not the result of the work of the author.)

Since binary executables are quite special, the author investigated several possible ways of improving slicing in a binary-specific manner. The improvements include static approaches that reduce the number of data dependence and summary edges in the dependence graphs as well as a dynamic approach that removes edges from the static call graph using dynamically collected information. These improvements are the joint work of the author and his co-authors, and the contribution of the author is the following: the author designed the lattice used for the improved stack access analysis, the author participated in the design of the dynamic improvement approach as well as in the implementation of the prototype slicer tool used to compute experimental results.

Code Obfuscation via Control Flow Flattening

The main result in the third part of the thesis is the adaptation of the code obfuscation method entitled control flow flattening to the C++ language. Based on paper [73], this result is presented in Chapter 13.

4. Control Flow Flattening of C++ Programs

To make control flow flattening of C++ programs possible, the author identified those constructs of the language that are not trivial to handle and gave solutions for them. Moreover, the author also designed an algorithm that can flatten functions of the C++ language and he gave its formal description. The author, jointly with his co-author, took part in the implementation of a prototype obfuscator tool and experimented with it in order to evaluate the

effect of control flow flattening on code comprehensibility. The experiments were also used to evaluate the suitability of source-to-source transformations for binary code obfuscation.

Chapter 16

Related Work

16.1 Program Slicing

Program slicing was introduced by Mark Weiser in 1979 as a static program analysis and extraction technique [102]. Weiser originally had many applications of slicing in mind. Most of these and many others were developed in the literature which followed. One of the primary initial goals of slicing was to assist with debugging. Weiser noticed [103] that programmers naturally form program slices, mentally, when they debug and understand programs. Therefore, it seemed natural to attempt to automate this process to improve the efficiency of the debugging process. Lyle and Weiser [80] further developed the theme of slicing as an aid to debugging and this remained a primary application of slicing for some time.

In this initial work on slicing, the algorithms used for slicing were based upon data flow equations [104]. However, in 1984 Ottenstein and Ottenstein [87] showed how the Program Dependence Graph (PDG) could be used to turn slicing into a graph reachability question. Ottenstein and Ottenstein's formulation was an intraprocedural one, however, and it was not clear how it could handle the calling context problem using the PDG. In 1998, Horwitz, Reps and Binkley [59, 60] introduced the System Dependence Graph (SDG), an extension of the PDG which could allow for the efficient computation of interprocedural slices while respecting calling context. Since then, the major-

ity of slicing algorithms have been SDG-based including those implemented in Grammatech's commercial program slicing tool, CodeSurfer [47].

The slices produced by the algorithm of Horwitz et al. are not necessarily executable programs [59]. The problem arises when different calling contexts require different subsets of a procedure's input parameters. Horwitz et al. propose two methods to transform non-executable SDG slices into executable programs. The first creates a copy of a procedure for each calling context requiring different subsets of the input parameters. The second option, later refined by Binkley [15], iteratively includes in*tra*procedural slices taken with respect to actual parameters until all calls to a procedure include the same parameters. This approach yields static slices that also satisfy the KL requirement of being execution path preserving.

In 1988 Korel and Laski [66] observed that slices might be more useful as a debugging aid if they could be constructed dynamically, taking into account the execution characteristics which led to the observation of erroneous behaviour. If slices are constructed dynamically, then they are guaranteed to be no larger than their static counter parts and they may be smaller. Korel and Laski's algorithm for constructing dynamic slices was a modified version of Weiser's data flow equations.

In 1990 Agrawal and Horgan [3] introduced two algorithms for constructing dynamic slices based on the PDG. (They actually proposed four algorithms, but two only impact performance and not the slices computed.) These two algorithms differ in ways made clear with the benefit of the theory introduced herein. In terms of the equivalence relations from Chapter 4, Agrawal and Horgan's first algorithm preserves $\mathcal{D}(\sigma, V, n)$ while their second algorithm and that of Korel and Laski both preserve $\mathcal{D}_{KLi}(\sigma, V, n^{(k)})$.

De Lucia et al. [36, 21] introduced a concept called conditioned slicing. The conditioned slicing criterion augments the traditional static criterion with a condition. The slicing process only needs to preserve the effect of the original program on the variables of interest if the condition is satisfied. By choosing this condition to be simply the constant predicate 'true', the definition of conditioned slicing becomes that of static slicing and by making it a conjunction of equalities, it is possible to mimic the effect of an input

sequence.

These observation led several authors to observe that conditioned slicing "subsumes" static and dynamic slicing [21, 34, 41]. However, this use of the term "subsumes" differs from the one used herein. It is based on the expressive power of the slicing criterion. The subsumption relations introduced herein are based on the semantics preserved by slicing and the set of programs that qualify as slices. Informally, it appears that conditioned slicing subsumes static slicing and is subsumed by dynamic slicing. The same is true for the iteration aware and execution path preserving variants.

Later theoretical works attempted to lay the foundations of slicing. However, these earlier works were primarily concerned with static slicing. Reps and Yang [91] showed that the PDG is adequate as a representation of program semantics, allowing it to be used in slicing and related program analyses without the loss of semantic information. Reps [89] showed how interprocedural-slicing can be formulated as a graph reachability problem. Cartwright and Felleisen [24] showed that the PDG semantics is a lazy semantics because of the demand driven nature of the representation, while Giacobazzi and Mastroeni [45] presented a transfinite semantics to attempt to capture the behaviour of static slicing. Harman et al. showed that slicing is lazy in the presence of errors [55]. Weiser [102] observed that his slicing algorithm was not dataflow minimal and speculated on the question of whether dataflow minimal slices were computable. Danicic showed how this problem could be reformulated as a theorem about unfolding [33], while Laurence et al. [74] showed how the problem can be expressed in terms of program schematology.

However, all this work was just concerned with static slicing. And to date very little formal theoretical analysis has been done on the properties of dynamic slicing. The closest prior work to that in the present thesis is the earlier work of Venkatesh [100] and the work by Harman et al. [49]. Venkatesh defined three orthogonal slicing dimensions, each of which offered a boolean choice. A slice could be static or dynamic, it could be constructed in a forward or backward direction and it could be either an executable program or merely a set of statements related to the slicing criterion. Venkatesh

therefore considered 2^3 slicing criteria, some of which had not, at the time, been thought of before (for example, the forward dynamic slice). Harman et al. introduced projection theory to analyse dynamic slicing. However, they used this theory to explain the difference between syntax-preserving and amorphous slicing, without addressing the issue of dynamic slicing.

Venkatesh also provided a formal description of program slicing. His semantic description was cast in terms of a novel denotational description of a labelled structured language using a concept of contamination. The idea was to capture the set of labels that identified statements and predicates whose computation would become contaminated when some particular variable was initially contaminated. Contamination propagates through the semantic description of a program in much the same way as data dependence and control dependence propagation is represented by the edges in a PDG [40, 60].

Venkatesh's approach does allow for a formal statement of the way in which dynamic and static slicing are related. However, Venkatesh was concerned with the three broad parameters of slicing and not with the details of dynamic slicing. As a result, he did not take account of the additional components of the dynamic slicing criterion: path preservation and iteration count sensitivity. Rather, Venkatesh's work was only cornered with Agrawal and Horgan's version of dynamic slicing and so, it avoided a lot of the subtlety found in the present work.

in the literature there are several surveys on slicing: Tip [98], and Binkley and Gallagher [17] provide surveys of program slicing techniques and applications. De Lucia [35] presents a shorter but more up-to-date survey of slicing paradigms. Binkley and Harman [18] present a survey of empirical results on program slicing. These papers provide a broad picture of slicing technology, tools, applications, definitions, and theory. By contrast, the first part of the present thesis is solely concerned with the formalisation and analysis of dynamic slicing.

16.2 Analysis and Slicing of Binary Code

The slicing of binary executables requires the building of a CFG from raw binary data. Debray et al. built a CFG for binaries compiled for the Alpha architecture in their code compaction solution [37], making use of a technique similar to the one outlined above.

To our knowledge, there are currently no practical interprocedural slicing solutions available for binary executable programs, and a useful intraprocedural binary slicing technique is also hard to find in the literature. Larus and Schnarr used an intraprocedural static slicing technique in their binary executable editing library called EEL [72]. They utilised slicing to improve the precision of control flow analysis, with indirect jumps mainly occurring in the compiled form of case statements. With the help of backward slicing, they were able to analyse constructs like these in an architecture and compiler-independent way.

Cifuentes and Fraboulet also made use of intraprocedural slicing for solving indirect jumps and function calls in their binary translation framework [27]. Bergeron et al. recommended using interprocedural static slicing for analysing binary code to detect malicious behaviour [8]. The computed slices should be verified against behavioural specifications in order to statically detect potentially malicious code. However, they did not elaborate on the potential problems of analysing binary executables, nor did they present any experimental results.

Antoniol et al. examined the static points-to analysis of C programs and investigated the impact of function pointers on the call graph [4]. Mock et al. analysed the feasibility of improving static slicing with dynamically gathered points-to data [83]. They carried out their experiments on C language sources and concluded that the information obtained might be especially useful in cases where function pointers are present.

16.3 Code Obfuscation

The scientific literature on program obfuscation is now about ten years old. A significant paper was written by Collberg, Thomborson, and Low [30], which describes the importance of obfuscation and summarises the most important techniques, mainly for the Java language. They gave a classification of the above-mentioned techniques and defined a formal method to measure their quality. In a later work [32], they focused on the obfuscation of the control flow of Java systems by inserting irrelevant but opaque predicates in the code. In their paper, they demonstrated that this method can give effective protection from automatic deobfuscators, while not increasing code size and runtime significantly. In another paper [31], they described a way of transforming data structures in Java programs. A summary of their results is given by Low [79], and a Java-targeted implementation is presented as well.

Similar to Collberg et al., Sarmenta studied parameterised obfuscators [92]. The parameters can select either the parts of the program where transformations shall be applied or the transformations to be applied. In addition, the transformations themselves can have parameters. Sarmenta investigated the combination of encryption and obfuscation as well. E.g., encrypted functions can be obfuscated or encryption can be performed during obfuscation.

In his PhD thesis, Wroblewski discussed low (assembly) level obfuscation techniques [106]. In his work, he analysed and compared the main algorithms of the field, and based on the results, he gave the description of a new algorithm. Zhuang et al. developed a hardware-assisted technique [109] which obfuscates the control flow information by dynamically changing memory accesses on-the-fly, thus preventing recurrent instruction sequences from being identified. Ge et al. presented another dynamic approach [44] where control flow obfuscation was based on a two-process model: control flow information is removed from the obfuscated program, and a concurrent monitor process is created to contain this information. During the execution of the program process, it continuously queries the monitor process, thus following the original path of control.

Wang et al. described an obfuscation technique [101] which combines

several algorithms, e.g., data flow transformation and control flow flattening. They showed that the problem of analysing and reverse engineering the code obfuscated by applying their technique is NP-complete. Chow et al. investigated control flow flattening [26] too, but they claimed that their approach worked for programs containing only simple variables, operators and labelled statements.

Code obfuscation is not only discussed in scientific papers, but it is utilised in several open source and commercial tools. Most of these tools are targeted for Java and work on byte code, e.g., Zelix Klassmaster [108], yGuard [107] and Smokescreen [77]. These tools perform name obfuscation (renaming of classes, methods and fields), encode string constants, and transform loops using gotos. The renaming technique is used by the Thicket tool family [93] and COBF [5] as well. Thicket supports several programming languages, while COBF is the only C/C++ obfuscator that is freely available.

The latter tool is the only one that is comparable to our prototype obfuscator implementation. Even though it transforms the names of classes, functions and variables, and removes spaces and comments from the source (thus making the code unreadable for a human analyzer), this offers no protection against automatic deobfuscators. We evaluated COBF on our benchmark functions but, as expected, we observed no change in the McCabe metric after obfuscation. What is more, in some cases, the renamings applied by COBF caused compile time errors.

Bibliography

[1] H. Agrawal. On slicing programs with jump statements. In *Proc. ACM SIGPLAN Conference on Programming Languages, Design and Implementation*, pages 302–312, June 1994.

[2] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software Practice and Experience*, 23(6):589–616, June 1993.

[3] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, New York, June 1990.

[4] G. Antoniol, F. Calzolari, and P. Tonella. Impact of function pointers on the call graph. In *Proc. of the 3rd European Conference on Software Maintenance and Reengineering (CSMR)*, pages 51–59, March 1999.

[5] Bernhard Baier. COBF. http://home.arcor.de/bernhard.baier/cobf/.

[6] T. Ball and S. Horwitz. Slicing program with arbitrary control-flow. In *Proc. International Workshop on Automated and Algorithmic Debugging*, pages 206–222, May 1993.

[7] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *IEEE/ACM* 15^{th} *Conference on Software Engineering (ICSE'93)*, pages 509–518. IEEE Computer Society Press, Los Alamitos, California, USA, 1993.

[8] Jean Bergeron, Mourad Debbabi, Mourad M. Erhioui, and Béchir Ktari. Static analysis of binary code to isolate malicious behaviors. In *Proc. IEEE International Workshop on Enterprise Security*, June 1999.

[9] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, March 2001.

[10] James M. Bieman and Linda M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

[11] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. Minimal slicing and the relationships between forms of slicing. In *Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005)*, pages 45–54, Budapest, Hungary, September 30 – October 1, 2005. IEEE Computer Society. Best paper award.

[12] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Lahcen Ouarbya. Formalizing executable dynamic and forward slicing. In *Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 43–52, Chicago, Illinois, USA, September 15–16, 2004. IEEE Computer Society.

[13] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. A formalisation of the relationship between forms of program slicing. *Science of Computer Programming*, 62(3):228–252, October 2006.

[14] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. Theoretical foundations of dynamic

program slicing. *Theoretical Computer Science*, 360(1–3):23–41, August 2006.

[15] David Wendell Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 3(1-4):31–45, 1993.

[16] David Wendell Binkley. The application of program slicing to regression testing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 583–594. Elsevier, 1998.

[17] David Wendell Binkley and Keith Brian Gallagher. Program slicing. In Marvin Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[18] David Wendell Binkley and Mark Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[19] David Wendell Binkley, Susan Horwitz, and Tom Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, 1995.

[20] David M. Blei. LDA-C. http://www.cs.princeton.edu/∼blei/lda-c/.

[21] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[22] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and G. A. Di Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)*, pages 424–433, Victoria, Canada, September 1994. IEEE Computer Society Press, Los Alamitos, California, USA.

[23] Gerardo Canfora, Aniello Cimitile, and Malcolm Munro. RE^2: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance: Research and Practice*, 6(2):53–72, 1994.

[24] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–27, 1989.

[25] J. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Trans. Program. Lang. Syst.*, 16(4):1097–1113, July 1994.

[26] 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.

[27] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural static slicing of binary executables. In *Proc. International Conference on Software Maintenance*, pages 188–195, October 1997.

[28] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'95)*, pages 124–133, Nice, France, 1995. IEEE Computer Society Press, Los Alamitos, California, USA.

[29] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. A specification driven slicing process for identifying reusable functions. *Software maintenance: Research and Practice*, 8:145–178, 1996.

[30] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, July 1997.

[31] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the IEEE International Conference on Computer Languages (ICCL'98)*, pages 28–38, Chicago, IL, May 1998.

[32] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL98)*, pages 184–196, San Diego, CA, January 1998.

[33] Sebastian Danicic. *Dataflow Minimal Slicing*. PhD thesis, University of North London, UK, School of Informatics, April 1999.

[34] Sebastian Danicic, Mohammed Daoudi, Chris Fox, Mark Harman, Robert Mark Hierons, John Howroyd, Lahcen Ouarbya, and Martin Ward. Consus: A lightweight program conditioner. *Journal of Systems and Software*, pages 241–262, 2005. Special Issue: Software Reverse Engineering.

[35] Andrea De Lucia. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[36] Andrea De Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, March 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[37] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, March 2000.

[38] Bruce Eckel. *Thinking in C++*, volume 1, chapter The C in C++, pages 125–126. Prentice Hall, 2nd edition, 2000.

[39] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus – reverse engineering tool and schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, 2002.

[40] 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*, 9(3):319–349, July 1987.

[41] Chris Fox, Sebastian Danicic, Mark Harman, and Robert Mark Hierons. ConSIT: a fully automated conditioned program slicer. *Software—Practice and Experience*, 34:15–46, 2004. Published online 26th November 2003.

[42] Keith B. Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[43] Keith Brian Gallagher. Evaluating the surgeon's assistant: Results of a pilot study. In *Proceedings of the International Conference on Software Maintenance*, pages 236–244. IEEE Computer Society Press, Los Alamitos, California, USA, November 1992.

[44] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. Control flow based obfuscation. In *DRM '05: Proceedings of the 5th ACM workshop on Digital rights management*, pages 83–92, New York, NY, USA, 2005. ACM Press.

[45] Roberto Giacobazzi and Isabella Mastroeni. Non–standard semantics for program slicing. *Higher-Order and Symbolic Computation*, 16(4):297–339, 2003. Special issue on Partial Evalution and Semantics-Based Program Manipulation.

[46] Rajiv Gopal. Dynamic program slicing based on dependence graphs. In *IEEE Conference on Software Maintenance*, pages 191–200, 1991.

[47] Grammatech Inc. The codesurfer slicing system, 2002.

[48] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 299–308, Orlando, Florida, USA, 1992. IEEE Computer Society Press, Los Alamitos, California, USA.

[49] Mark Harman, David Wendell Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, October 2003.

[50] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, 5(3):143–162, September 1995.

[51] Mark Harman and Sebastian Danicic. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[52] Mark Harman, Rob Mark Hierons, Sebastian Danicic, John Howroyd, and Chris Fox. Pre/post conditioned slicing. In *IEEE International Conference on Software Maintenance (ICSM'01)*, pages 138–147, Florence, Italy, November 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[53] Mark Harman and Robert Mark Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[54] Mark Harman, Lin Hu, Robert Mark Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, January 2004.

[55] Mark Harman, Dan Simpson, and Sebastian Danicic. Slicing programs in the presence of errors. *Formal Aspects of Computing*, 8(4):490–497, 1996.

[56] Robert Mark Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[57] Robert Mark Hierons, Mark Harman, Chris Fox, Lahcen Ouarbya, and Mohammed Daoudi. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability*, 12:23–28, March 2002.

[58] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[59] Susan Horwitz, Thomas Reps, and David Wendell Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[60] Susan Horwitz, Thomas Reps, and David Wendell Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[61] Mariam Kamkar. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993. Available as Linköping Studies in Science and Technology, Dissertations, Number 297.

[62] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In *Proceedings of the* 4^{th} *Conference on Programming Language Implementation and Logic Programming*, pages 370–384, 1992.

[63] Ákos Kiss, Judit Jász, and Tibor Gyimóthy. Using dynamic information in the interprocedural static slicing of binary executables. *Software Quality Journal*, 13(3):227–245, September 2005.

[64] Ákos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. Interprocedural static slicing of binary executables. In *Proceedings of the 3rd*

IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003), pages 118–127, Amsterdam, The Netherlands, September 26–27, 2003. IEEE Computer Society.

[65] Donald E. Knuth and Robert W. Floyd. Notes on avoiding "go to" statements. *Information Processing Letters*, 1(1):23–31, February 1971.

[66] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[67] Bogdan Korel and Jurgen Rilling. Dynamic program slicing in understanding of program execution. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)*, pages 80–89, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[68] Bogdan Korel and Jurgen Rilling. Dynamic program slicing methods. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 647–659. Elsevier, 1998.

[69] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, 2003.

[70] S. Kumar and S. Horwitz. Better slicing of programs with jumps and switches. In *Proc. FASE 2002: Fundamental Approaches to Software Engineering*, April 2002.

[71] Arun Lakhotia. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)*, pages 34–44, 1993.

[72] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. *ACM SIGPLAN Notices*, 30(6):291–300, June 1995.

[73] Tí mea László and Ákos Kiss. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientiarum de Rolando Eötvös*

Nominatae – Sectio Computatorica, XXX, 2008. Accepted for publication.

[74] Michael R. Laurence, Sebastian Danicic, Mark Harman, Rob Hierons, and John Howroyd. Equivalence of conservative, free, linear program schemas is decidable. *Theoretical Computer Science*, 290:831–862, January 2003.

[75] Keith Lea. Java is faster than C++ benchmark. http://www.kano.net/javabench.

[76] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. International Symposium on Microarchitecture*, pages 330–335, 1997.

[77] Lee Software. Smokescreen. http://www.leesw.com/smokescreen/obfuscation.html.

[78] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, July 1979.

[79] Douglas Low. Java control flow obfuscation. Master's thesis, Department of Computer Science, University of Auckland, June 1998.

[80] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications*, pages 877–882, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California, USA.

[81] Thomas J. McCabe and Arthur H. Watson. Software complexity. *Crosstalk, Journal of Defense Software Engineering*, 7(12):5–9, December 1994.

[82] Microsoft Corporation. Microsoft Portable Executable and Common Object File Format specification version 6.0, February 1999. http://www.microsoft.com/hwdev/hardware/PECOFF.asp.

[83] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Improving program slicing with dynamic points-to data. In *Proc. 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 71–80, November 2002.

[84] Steven S. Muchnick. *Advanced Compiler Design & Implementation*, chapter Approaches to Control-Flow Analysis, pages 172–177. Morgan Kaufmann Publishers, 1997.

[85] Netlib. Linpack benchmark. http://www.netlib.org/benchmark.

[86] Linda M. Ott and Jeff J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81, Baltimore, Maryland, USA, May 1993. IEEE Computer Society Press, Los Alamitos, California, USA.

[87] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184, 1984.

[88] Lyle Ramshaw. Eliminating goto's while preserving program structure. *Journal of the ACM*, 35(4):893–920, 1988.

[89] Thomas Reps. Program analysis via graph reachability. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 701–726. Elsevier Science B. V., 1998.

[90] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[91] Thomas Reps and Wuu Yang. The semantics of program slicing. Technical Report Technical Report 777, University of Wisconsin, 1988.

[92] Luis F. G. Sarmenta. *Protecting programs from hostile environments : encrypted computation, obfuscation and other techniques*. PhD thesis,

MIT Department of Electrical Engineering and Computer Science, July 1999.

[93] Semantic Designs. Thicket family of source code obfuscators. http://www.semdesigns.com/Products/Obfuscators/index.html.

[94] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.

[95] Dan Simpson, Samuel H. Valentine, Richard Mitchell, Lulu Liu, and Rod Ellis. Recoup – Maintaining Fortran. *ACM Fortran forum*, 12(3):26–32, September 1993.

[96] Standard Performance Evaluation Corporation (SPEC). SPEC CINT2000 benchmarks, 2000. http://www.spec.org/osg/cpu2000/CINT2000/.

[97] Bjarne Stroustrup. *The C++ Programming Language*, chapter Expressions and Statements, page 141. Addison-Wesley, 3rd edition, 1997.

[98] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[99] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) version 1.2, May 1995. http://www.x86.org/ftp/manuals/tools/elf.pdf.

[100] Guda A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[101] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, May 2000.

[102] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[103] Mark Weiser. Programmers use slicing when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

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

[105] Mark Weiser and James R. Lyle. *Experiments on slicing–based debugging aids*, chapter 12, pages 187–197. Empirical studies of programmers, Soloway and Iyengar (eds.). Molex, 1985.

[106] Gregory Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Institute of Engineering Cybernetics, Wroclaw University of Technology, 2002.

[107] yWorks GmbH. yGuard. http://www.yworks.com/en/products_yguard_about.html.

[108] Zelix Pty Ltd. Zelix klassmaster. http://www.zelix.com/klassmaster/index.html.

[109] Xiaotong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, and Santosh Pande. Hardware assisted control flow obfuscation for embedded processors. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 292–302, New York, NY, USA, 2004. ACM Press.