Program Code Analysis and Manipulation

Summary of the 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

Introduction

"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 foundations of program slicing, the application of program slicing to binary programs, and the obfuscation of programs written in C++ language.

The author admits that 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 research topics of the author, the thesis – which is summarised in this booklet – has been titled *Program Code Analysis and Manipulation* to match the terminology used by the wider software engineering community.

In the thesis, the author states four main results which are listed below:

1. The Unified Framework of the Program Projection Theory

2. Analysis of the Relationships between Forms of Slicing

3. Dependence Graph-based Slicing of Binary Executables

4. Control Flow Flattening of C++ Programs

Henceforth, this summary follows the structure of the thesis i.e., it is composed of three main parts that reflect the research topics of the author. In the following sections, the three fields of code analysis and manipulation are briefly introduced and the above listed main results are presented with an emphasis on the author's own contribution to these results.

The Theory of Slicing

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 [25]. 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.

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 [9, 10], which was first used to examine the similarities and differences between the amorphous and the syntax-preserving forms of slicing, including Weiser's static slicing. This study uses projection theory to investigate the nature of dynamic slicing too, as originally formulated by Korel and Laski [14].

The Unified Framework

A common belief 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 1 makes clear, Korel and Laski's (KL) definition of dynamic slicing is incomparable with the definition of static slicing. In Figure 1, program q_1 is a valid static slice of p_1 with respect to $(\{\mathtt{y}\}, 7)$ but it is

`1   x=1;`	`1   x=1;`
`2   x=2;`	
`3   if (x>1)`	`3   if (x>1)`
`4     y=1;`	`4     y=1;`
`5   else`	`5   else`
`6     y=1;`	`6     y=1;`
`7   z=y;`	`7   z=y;`
p_1: Original Program	q_1: Slice w.r.t. $(\{\mathtt{y}\}, 7)$

Figure 1: A static slice, which is not a KL–slice.

not a KL–dynamic–slice with respect to $(\langle\rangle, 7^4, \{\mathtt{y}\})$, since having different execution paths violates the definition of KL–slicing. 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 the requirement on the execution path, it is also more strict.

The identification of the main cause of incomparability between static slicing and KL–slicing made us try to fit KL–slicing into the framework of projection theory. However, the hitherto [9, 10] used definitions could not capture the execution path requirement. Thus, we had to extend the existing definitions. This extension made us realise that there was another component hidden in the KL-slicing criterion: the iteration count. As a result, we decided to set up a unified framework based on a unified semantic equivalence relation which is capable of expressing Korel and Laski's dynamic slicing as well.

Definition 1 (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:

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

where $\overline{p}$ and $\overline{q}$ denote the set of statement numbers in p and q, and T_p^σ and T_q^σ denote the state trajectories resulting from the execution of p and q in σ, respectively. The definition of the auxiliary function Proj^* is left for the thesis.

In the above 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 'input' part of the slicing criteria. The set of variables of interest V is common to all slicing criteria. Parameter P 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.

By instantiating Definition 1 with appropriate parameters we get a new equivalence relation which captures the semantics of Korel and Laski's dynamic slicing.

Definition 2 (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:

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

In the thesis, we will also show that Definition 2 faithfully captures Korel and Laski's definition.

Theorem 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.*

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 3 (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, $\varepsilon(x, y) = \emptyset$.

Moreover, now that we have identified the orthogonal criterion components (set of initial states, 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 resulting from the other possible parameterisations of the unified equivalence. These equivalence relations are defined below, with the already presented relations repeated for the sake of completeness.

Definition 4 (Eight Equivalences)**.**

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

Six equivalence relations of the above eight capture the semantic property of six new, hitherto undiscussed slicing methods.

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. 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 (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

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

This subsumes relation is a partial ordering of parameterised equivalence relations. Figure 2 presents the lattice of the subsumes relation for $\mathcal{S}$, $\mathcal{S}_i$, $\mathcal{D}$, $\mathcal{D}_i$, $\mathcal{S}_{KL}$, $\mathcal{S}_{KLi}$, $\mathcal{D}_{KL}$ 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 [6, 8, 23]. In the thesis, we shall also demonstrate the correctness of the diagram in Figure 2:

Theorem 2. *The lattice shown in Figure 2 is correct: two parameterised equivalence relations are connected in the diagram if and only if they are in subsumes relation.*

Tn the above we studied the relationships between the *semantic* properties of eight forms of slicing. In general, however, we are interested in the relation between the forms of slicing. In order to achieve this, we will need to

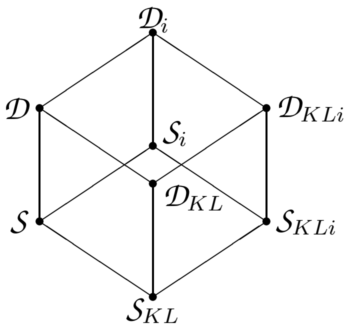
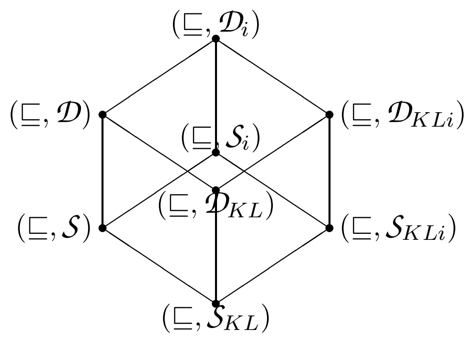


Figure 2: Subsumes relationship between equivalence relations.



$(\sqsubseteq, \mathcal{D}_i)$
$(\sqsubseteq, \mathcal{D})$
$(\sqsubseteq, \mathcal{D}_{KLi})$
$(\sqsubseteq, \mathcal{S}_i)$
$(\sqsubseteq, \mathcal{D}_{KL})$
$(\sqsubseteq, \mathcal{S})$
$(\sqsubseteq, \mathcal{S}_{KLi})$
$(\sqsubseteq, \mathcal{S}_{KL})$

Figure 3: Subsumes relationship between slicing techniques.

take account of both the syntactic ordering relation and the semantic equivalence relation. We call this combination a slicing technique, and we define subsumes relations between the slicing techniques as well.

The 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 helps prove the correctness of the diagram depicted in Figure 3, which shows the precise connection 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 $\mathcal{D}_{KLi}$.

These results on the relationship between the eight equivalence relations and the derived slicing techniques are both theoretically interesting and practically important. They allow slice users to understand and then choose the most appropriate slicing definition for a given problem.

Syntactic Ordering of Slicing Techniques

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. 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 formalise these views and be able to determine whether one definition of slicing leads to inherently smaller slices than another, we shall extend syntactic ordering notion and apply it to *slicing techniques* where the extension is based on the comparison of sets of minimal slices (since minimal slices are

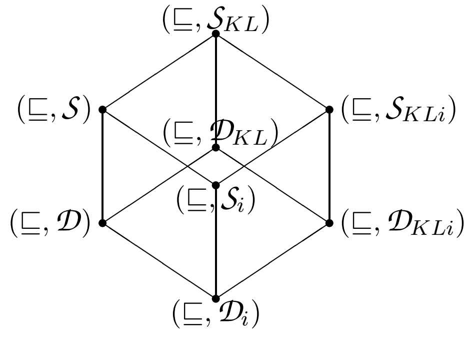

$(\sqsubseteq, \mathcal{S}_{KL})$
$(\sqsubseteq, \mathcal{S})$
$(\sqsubseteq, \mathcal{S}_{KLi})$
$(\sqsubseteq, \mathcal{D}_{KL})$
$(\sqsubseteq, \mathcal{S}_i)$
$(\sqsubseteq, \mathcal{D})$
$(\sqsubseteq, \mathcal{D}_{KLi})$
$(\sqsubseteq, \mathcal{D}_i)$

Figure 4: Slicing techniques ordered by traditional syntactic ordering.

not necessarily unique).

In the thesis, we state a theorem concerning 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. (Interestingly, the converse of this theorem does not hold.)

The above theorem, which is stated informally here, provides the basis for the comparison of slicing techniques. It provides the necessary machinery to show that a duality exists between subsumes relation and syntactic ordering over slicing techniques.

Theorem 3 (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 $\lesssim$,*

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

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.

Although the converse of Theorem 3 is not true in general, we can state stronger results for the eight slicing techniques obtained by combining the traditional syntactic ordering $\sqsubseteq$ and the eight equivalences given in Definition 4. They form a lattice which is isomorphic (in this case inverted) to that given in Figure 3. This is shown in Figure 4.

Theorem 3 tells us that whenever two slicing techniques are related as in Figure 3, i.e., they are in subsumes relation, then they have an inverse syntactic ordering relationship. That is, in the “if” direction, the correctness of Figure 4 is proven. However, $(\lesssim, \approx_A) \not\subseteq (\lesssim, \approx_B)$ does not imply that

$(\lesssim, \approx_B) \not\lesssim (\lesssim, \approx_A)$; thus, it must be shown that the slicing techniques not related in Figure 4 are really not related according to the traditional syntactic ordering. This is stated by the following theorem (and is proven in the thesis).

Theorem 4 (Duality of the Eight Forms of Slicing (only if))**.** *If two slicing techniques are not connected in Figure 4, then they are not related according to the traditional syntactic ordering.*

The above results establish 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 syntactic ordering tells us which produces the best (i.e., smallest) slices.

Main Results and Own Contribution

1. The Unified Framework of the 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

Since the introduction of Weiser's original concept, several slicing algorithms have been proposed [19, 14, 11, 6, 9, 1]. These algorithms were originally developed for slicing high-level structured programs. 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 [7] presented a technique for the intraprocedural slicing of binary executables, but we are not aware of any usable interprocedural solution.

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, viruses and post-link time modified programs. (These include source code recovery, binary bug fixing and code transformation.)

Control Flow Analysis

Many tasks in the area of code analysis and manipulation require a control flow graph (CFG). It is also necessary for program slicing to have a CFG of the sliced program. However, the control flow analysis of a binary executable has a number of associated problems.

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 boundaries of the low-level instructions have to be detected. On architectures with *variable length instructions*, the boundaries may not be detected unambiguously. On other architectures where *multiple 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. If the binary representation *mixes code and data*, as is typical for most widespread architectures, their separation has to be carried out as well.

Provided that we have identified the instructions, we may begin to build the nodes of the graph. First, the *basic blocks* need to be determined using *basic block leader* information. 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 *exit node* is created to represent its single exit point.

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. The correct detection of the possible control transfers requires the behaviour analysis of machine instructions. Even the high number of instruction types may be hard to cope with, but the hardest problem arises with those control transfer instructions where the *target cannot be determined unambiguously.* To correctly handle these instructions, two new CFG node types have to be introduced, the *unknown function* and *unknown block* nodes, which represent the targets of indirect calls and jumps, respectively. These nodes are linked to all the possible targets of the indirect control transfers.

Another source of problems is when control is transferred between functions in a way that is different from a function call. *Overlapping* and *cross-jumping* functions are typical examples of this problem. With these constructs, the exit node of the control transferring function is not reached. To compensate for this, a control flow edge has to be inserted between the exit nodes of the affected functions.

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 identify the potential targets of indirect jumps and calls? Fortunately, most executable file formats [22, 17] can store extra symbolic and relocation information along with the raw binary data that may be employed to separate code and data in the binary image, assist in detecting function boundaries and instruction set switches, or help in determining the targets of ambiguous control transfers. Our experiences with several tool chains and file formats have shown that with an appropriate specification, the necessary information can be retrieved relatively easily.

Dependence Graph-based Slicing

Once the interprocedural control flow graph is built, we perform a control and data dependence analysis for each function found in the CFG, which results in a program dependence graph (PDG).

During data dependence analysis, the inherent differences between high-level languages and low-level instructions have to be handled. 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. We analyse each instruction to determine which registers and flags it reads and writes. In addition, the memory access of the instructions has to be analysed as well. A conservative approach is just to find out whether an instruction reads from or writes to the memory. Moreover, 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.

The PDGs built so far can be used to compute intraprocedural slices just by traversing the graph via control and data dependence edges [19]. However, by interconnecting the PDGs with *parameter-in* and *parameter-out* edges, and by adding *summary* edges [20] as well, we create the system dependence graph (SDG) of the program. The SDG built this way can be used to compute interprocedural slices using the two-pass algorithm of Horwitz et al. [11].

Improving the Slicing

Although the dependence graphs built as described above are safe, they are overly conservative. One way of improving them is by using architecture specific information. 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. If the set of saved and restored registers can be determined, we can reduce the set of output parameters of the functions.

Another approach is to improve the conservative handling of memory accesses in data dependence analysis. On most architectures the number of available registers is limited. Thus, values and results that cannot be assigned to registers are usually stored in the stack. However, the memory model outlined above is very simple, so data dependence analysis cannot accurately detect the dependences across the stack. As a solution to this problem, we propose an improved memory model, which is based on the characterisation of the registers at each instruction of the program with a pair of lattice elements to represent information about their contents at the entry and exit points of the instruction. The lattice and its elements are shown in Figure 5. (The lattice element $\perp$ tells us that whether the register contains a reference in the stack or not cannot be statically determined. Assigning M to a register means 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.)

The fix-point iteration algorithm used to propagate these lattice elements through the control flow graph is elaborated on in the thesis. Using the results of this process, data dependence analysis can be improved so as to avoid adding superfluous dependence edges to the graph.

A third way of improving the slicing is based on our observation that the high number of unresolved indirect function calls often results in too large slices. We followed the idea of Mock et al. [18] of using *dynamic* points-to

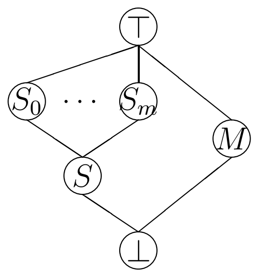


Figure 5: The lattice to characterise register content.

information in static slicing. More precisely, we gather dynamic information at each statically unresolved indirect call site. If the application-to-be-sliced is executed in a controlled environment on some representative input, it is possible to determine the realised targets of the statically unresolved indirect call sites and thus, to replace call edges to the unknown function node with call edges to the actual targets. Experiments show that the use of dynamic information can result in a huge reduction in the number of call edges. Although the resulting call graphs may be imprecise and the slices may become unsafe, in some situations this limitation is acceptable.

Experimental Results with Static Slicing

We implemented a slicer for statically linked binary ARM executables and evaluated it on programs taken from various benchmark suites. The size of code in the executables ranged from 12 to 419 kilobytes.

First, we built the CFG for all the selected programs, as described above. Once the CFGs were present, we performed control and data dependence analyses (both the conservative and statically improved ones) to obtain PDGs for each reachable function, and finally, we created the SDGs. The static improvements yielded a 28% and 51% reduction in the number of data dependence and summary edges on average, 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 the 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). We obtained slices 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.

There are situations (e.g., programs modified at post-link time) where library code also becomes important. For this reason we computed slices for those (reachable) functions as well which originate from library code. The results show trends similar to those above. The slices computed using the 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 slices is the high number of statically unresolved function calls.

Experimental Results with Dynamic Improvements

To gather dynamic information about the selected benchmark programs, we executed them in the emulator of Texas Instruments. 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. As expected, the number of call edges is significantly reduced in those applications which make intensive use of indirect function calls. Even those programs that contained only a few indirect call sites and indirectly callable functions showed a clear reduction.

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 dynamically improved one. 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. 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. Those programs that use no indirect function calls, not surprisingly, brought no improvements. Two programs using indirect function calls only rarely achieved a 6% reduction. However, in the case of two programs which make intensive use of indirect function calls, the average size of the slices computed using the dynamically improved call graph fell by 72% and 57%, respectively, compared to the static approach.

Main Results and Own Contribution

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

Protecting programs from unauthorised access has always been a concern of software vendors. The goal is usually to make the job of the attacker as difficult as possible. Here, 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.

Although several large software systems are still written in C++, to date only a few tools have been designed specifically for their protection. Since the importance of protecting C++ programs is not negligible, here we will set out the goal to develop obfuscation techniques for C++.

Importantly, we are interested in the effect of the algorithms not just on the source code level, but on the binary level as well since lots of attacks are directed against programs released in binary form to work around or to deactivate their protection. Thus, we investigated whether a static source-to-source transformation can render the comprehension of the binary code more difficult as well.

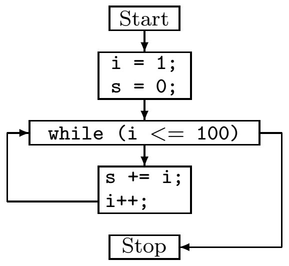
Flattening of C++ Programs

Here, we discuss the adaptation of control flow flattening [24] to the C++ language. The idea behind the technique 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 to 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 `switch` statement with each block in a separate case, and the `switch` 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 `switch`. An example of this method is given in Figure 6.

According to this description, the task of flattening a function seems to be quite simple. However, when it comes to the application of the idea to a real programming language, then we run into certain problems.

As the example given in Figure 6 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



(a)

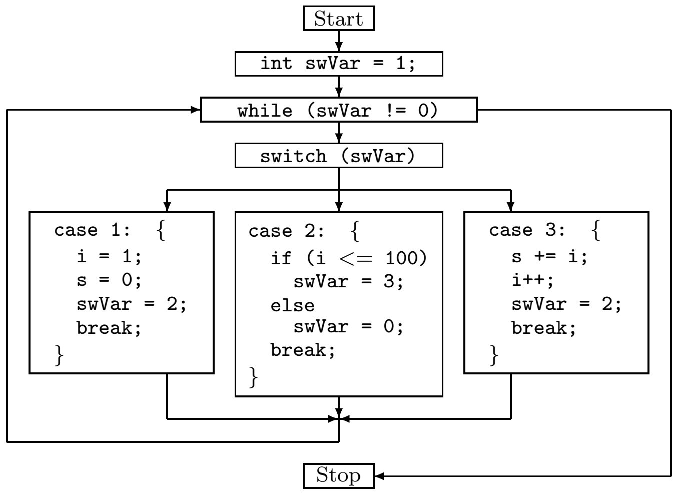

Start
int swVar = 1;
while (swVar != 0)
switch (swVar)
case 1: { i = 1; s = 0; swVar = 2; break; }
case 2: { if (i <= 100) swVar = 3; else swVar = 0; break; }
case 3: { s += i; i++; swVar = 2; break; }
Stop
(b)

Figure 6: The effect of control flow flattening on the control flow graph (a: original, b: flattened).

its body detached definitely cannot reproduce the original behaviour.

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` is 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 [21].)

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` they were intended to do, they would terminate or restart the control structure of the flattened code.

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 handling mechanism, and thrown exceptions could not be caught by the originally intended handlers.

In the thesis, not only examples are shown to help solve the above-listed problems, but the formal description of an algorithm is given as well that is designed to flatten C++ functions.

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 collected a benchmark which consisted of 23 functions. These functions were obfuscated using a prototype tool which implements the obfuscation technique introduced in the thesis. Before and after obfuscation, we computed McCabe's cyclomatic complexity metric [16] from the source representation of each function to measure the change in their complexity and comprehensibility. Afterwards, we compiled both the original and the obfuscated codes to ARM target (both with compiler optimisations switched off and on). The resulting binaries were analysed using the same tool that we applied for slicing binary executables. We used this tool to compute McCabe's metric for the original and obfuscated codes and this data was in turn used to measure the change in the complexity and comprehensibility of the binary programs.

By investigating the changes in McCabe's metric we found a significant, 4.63-fold increase in the complexity of the source code on average. Moreover, the measurements obtained support our assumption that the 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 non-optimised and optimised binaries, respectively. The somewhat smaller increase in the case of optimised binaries can be attributed to the strong optimisation techniques applied by the compiler. 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.

Our analysis of the data shows that the effect of the algorithm on complexity is linearly proportional to the original complexity. For source code and for both binary versions, Figure 7 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. The results 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 non-optimised and optimised binary code is only 1.55-fold and 1.57-fold on average. In addition, we counted the number of executed instructions as well. We found that the increase in the number

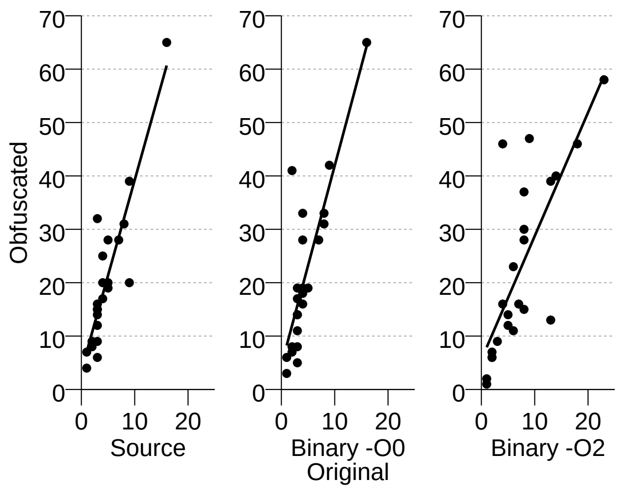


Figure 7: The relationship between the complexities of the original and the flattened code.

of executed instructions is 2.03-fold and 2.39-fold on average for the non-optimised and 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, which means that in real applications both the static and the dynamic effects on the resource consumption of the whole program should be much smaller.)

Main Results and Own Contribution

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.

Summary

The author presented three areas of the domain of program code analysis and manipulation: the theoretical foundations of program slicing, the application of program slicing to binary programs, and the obfuscation of programs written in C++ language. These three are discussed below.

Being interested in the formal definitions of slicing, the author decided to compare Weiser's static slicing and Korel and Laski's dynamic slicing. Based on the results, the author found it necessary to create a unified framework of program projection theory. The created framework not only made it possible to re-define existing and well-known slicing techniques, but it also led to the identification of six new possible forms of slicing that were hitherto unknown in the literature. Moreover, the author defined a subsumption relationship of forms of slicing and then he showed that eight slicing techniques form a lattice. The author also found that slicing techniques can be ordered based on sets of minimal slices and that the resultant ordering is the dual of the subsumption relationship. These results are both theoretically interesting and practically important, since they allow slice users to find the most appropriate slicing definition for a given problem.

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. The lack of existing solutions is hard to understand since there are special applications of the slicing of programs without source code. Thus, as dependence graph-based slicing requires a control flow graph, the author decided to explore the problems of control flow analysis of binary programs and proposed solutions as well. Moreover, since binary executables are quite special, the author investigated several possible ways of improving slicing in a binary-specific manner, including both static and dynamic approaches. The experiments were carried out using a prototype slicer tool.

Since the importance of protecting C++ programs is quite important, the author set himself the goal of developing obfuscation techniques for C++, which can act as a first line of defence. The author also discussed the adaptation of a technique called control flow flattening to C++. The author identified those constructs of the language that are not trivial to handle and found solutions for them. In addition, the author designed an algorithm that can flatten functions written in the C++ language. The technique was implementated in a prototype obfuscator tool and both its effects on the source and on the binary code were carefully evaluated.

Lastly, Table 1 summarises which publications cover which results of the thesis.

	[3]	[2]	[5]	[4]	[13]	[12]	[15]
1.	●		●				
2.		●	●	●			
3.					●	●	
4.							●

Table 1: Relation between the main results of the thesis and the corresponding publications.

Acknowledgements

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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

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

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

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