Dependence-based static program slicing and its approximations

Summary of the Ph.D. Dissertation

by **Judit Jász**
Supervisor: Dr. Tibor Gyimóthy

Ph.D. School of Computer Science

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

Szeged
2009

Introduction

The activity of program slicing is similar to what an experimental programmer does during debugging in order to understand the behavior of the program at a particular point in it. To achieve his aim, he divides the program into smaller parts and identifies the statements which determine the behavior of the given program point. This reduced program is called a program slice, which is executable, and its semantics at the given program point is the same as the semantics of the full program. Formally speaking, a program slice is the set of statements which potentially affects the behavior of a given program point, called a *slicing criterion*. This original definition of program slicing was introduced by Mark Weiser in 1979 [26].

Since the introduction of the original concept of slicing, various notions of program slices have been proposed [5; 6; 18; 25]. The main reason for these differences is the fact that different applications of program slicing require slices with different features. Many papers and surveys in the field of software maintenance describe the definitions and modifications of program slices and present new applications based on the modified definitions. The original definition of a program slice provides one of the greatest classes of program slicing, which is referred to as a *backward slice*. It is backward because, starting from a given program point, it identifies the events and statements in the program which may be execute before the given point and may be responsible for its bad behavior.

Of course, it is also possible that an application based on program slicing has a totally different motivation. If one wants to investigate the effect of a program point in a program, *forward slicing* can be of help for reducing the size of the set of statements which are affected by the given program point. If one examines the program in a concrete execution or wishes to determine the general behavior of the program at the given program point, we can apply techniques of *dynamic* and *static program slicing*, respectively.

In the thesis we focus on static program slicing and one of its potentially applications, namely an examination of the dependencies inside the program. In the first part we use the dependence graphs and dependence relations to determine the static program slices of binary executables, then in the second part we apply the results of program slicing to validate the dependencies determined via a method distinct from program slicing. The topics of the thesis can be put into two main groups.

I/1 **Problems and solutions during the determination of the dependence graphs of binary programs. The adaptation of dependence-based program slicing to binary executables.**

I/2 **Improved slicing algorithms for binary executables.**

I/3 **Experimental results of the static program slicing of binary executables.**

II/1 **Defining Static Execute After (SEA) and Static Execute Before (SEB) relations and a suitable program representation for finding these relations. Algorithms for computing the SEA and SEB relations.**

II/2 **Experimental comparison of SEA and SEB relations with relations computed by program slicing.**

II/3 **Hidden dependencies in object-oriented programs; experimental investigation of SEA and SEB relation in object-oriented programs.**

In the following sections, I briefly describe the above results and I emphasize my own contributions in these at the end of each section.

Program slicing of binary executables

Although the slicing of programs written in a high-level language has been extensively studied in the literature, very few papers have addressed the issue of the slicing of binary executable programs. The lack of existing solutions is really hard to understand since the application domain for slicing binaries is similar to that for slicing high-level languages. The program slicing of binary executables can be applied to understand the behavior of programs without source code like assembly programs, legacy software, commercial off-the-shelf (COTS) products, viruses and post-link time modified programs. Although some papers deal with the intraprocedural program slicing of binary executables [9; 20], there are only suggestions about how to use dependence graph-based interprocedural slicing to analyze binaries [2], but these papers do not discuss the handling of the problems that arise or provide any concrete experimental results. Since binaries have many features which are not present in high level languages, the methods devised for high level languages generally cannot be adapted to binaries.

I/1. Problems and solutions during the determination of the dependence graphs of binary programs. The adaptation of dependence-based program slicing to binary executables.

A potential method for determining the interprocedural slices of a program is defining the dependence graphs of the program, and with an appropriate traversal we get the wanted slices [12]. During the building of the dependence graphs of binary programs many problems exist which are not present in high level languages.

Since the *control flow graph – (CFG)* is needed in many other applications like code analysis, code generation and code compaction, there are many papers which deal with this topic [10; 15]. Depending on the architecture employed, different problems arise during the building of the dependence graphs. The determination of the control flow graphs in the case of binary programs includes not only the determination of the control flow edges among statements, but the determination of statements and function boundaries as well.

After the determination of the control flow information, we have to compute the *data dependence graph – (DDG)* and the *control dependence graph – (CDG)*, which together constitute the *program dependence graph – (PDG)*. Extending the PDG with the appropriate dependence edges, we get the *system dependence graph – (SDG)* which is applied to compute the interprocedural slice of the slicing criterion.

The determination of control dependencies with the control flow information is quite easy, but we have to handle the overlappings and the cross-jumping functions. Since binary executables can transfer control to another function in a way other than the function call, before the computation of the dominance relations we have to extend the program points of a function with each statements, that is reachable without a function call from the entry point of the given function.

For binary executables the most difficult task is the determination of the data dependencies. 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 conservative approach, we only determine whether a given statement reads or writes any memory location, thus representing the whole memory as only one argument.

Unlike that for high-level programs, in binaries the parameter list of procedures is not explicitly defined, but it has to be found via a suitable interprocedural analysis. We introduced a fix-point iteration method like that shown in Figure 1 to determine the

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

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

Figure 1: Determination of the U_f and D_f sets, where U_f and D_f sets are the used and defined elements of the function f respectively. I_f is the set of instructions of f, C_f is the set functions called by f, and lastly u_j and d_j sets are the used and defined elements (registers, flags and memory locations) of the instructions j.

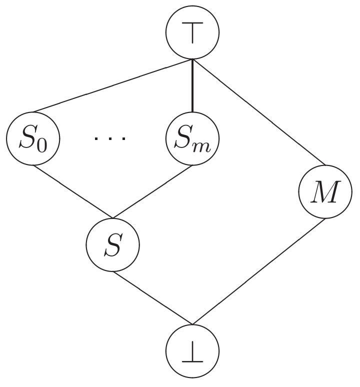
used and defined arguments of each function.

We have to augment the graph representation of the binary program with appropriate nodes which represent the used or defined registers, flags and memory locations, formal input and output parameters, and the actual input and output parameters. Next the data dependence edges can be defined via a traversal of the control flow graph. The summary edges needed for interprocedural slicing may be computed after implementing the appropriate algorithm [22].

I/2. Improved slicing algorithms for binary executables.

Although the computation of the dependence graphs is safe, because we do not ignore any existing dependencies, it is too conservative due to the conservative approach of the data dependence analysis and the lack of architecture specific information use. We can improve both the data dependencies and the control dependencies either by refining the static analysis or with the help of some dynamically gathered information. In the thesis we present two static approaches for improving the precision of the DDG and two dynamic approaches for refining the call graph. While the static approaches are safe, the dynamic approaches are imprecise in both cases, so the slices may become unsafe. In some situations, such as when we are debugging with limited resources, this approach is acceptable.

The first static approach is based on a heuristic analysis of function prologs and



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

Figure 2: The lattice and the meet operation of the lattice for improving the handling of memory use.

epilogs. In most current architectures, various function calling conventions exist which specify what portions of the register of a function have to be keep intact when called. Using this information we can reduce the number of defined registers for each function, and hence reduce the number of the summary edges and the size of the program slices.

In the second static approach, we attempted to refine the conservative handling of data dependencies using a more sophisticated analysis of the memory access of the instructions. At the binary level, the high-level concepts of variables and function parameters do not exist, so the compilers use registers in their place. But because in most architectures the number of available registers is limited, 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 part of the memory called the stack.

In our procedure, we mark all registers at a given instruction location by a pair of lattice elements to represent statically collected information about their contents at the entry and exit points of the instruction. 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 it cannot be statically determined whether the register contains a reference in the stack or not. Assigning M to a register means that it does not contain a reference in the stack. The lattice element S indicates that the register 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. Figure 2 above shows the lattice and its meet operation. After an appropriate initialization, a fix-point iteration algorithm is used to propagate these lattice elements through the control flow graph. The thesis describes this iteration algorithm in detail.

In the case of binary executables, especially in larger programs, there are many statically unresolved function calls. In these situations, the target of the call may be all the functions, whose addresses were used during the execution of the program. Needless to say, these situations introduce many unnecessary edges in the dependence graphs. With the dynamic improvements, we attempted to refine the dependencies caused by indirect function calls.

To enable the gathering of dynamic information, we need to determine the run-time address of each statically unresolved indirect call site when the construction of the CFG is completed and write each address to the disk. Then the application can be executed in a controlled environment on some representative input. The previously determined addresses are used as breakpoints where dumping the registers to a log file should be performed. With the help of the generated log files, it is possible to determine the realized targets of the statically unresolved indirect call sites.

The call sites which were not executed during any invocation of the application have no associated dynamic information so they can be handled in various ways. One is to retain the call edges where the call site was not covered by any of the dynamic executions. Another is to rely entirely on dynamic data and treat them as calling no functions, but this solution may result in over-optimistic slices.

I/3. Experimental results of the static program slicing of binary executables.

We implemented our solutions and evaluated them on programs taken from the SPEC CINT2000 [24] and Media Bench benchmark suites [21]. The selected programs were compiled using Texas Instruments' TMS470R1x Optimizing C Compiler version 1.27e for the ARM7T processor core with Thumb instruction set. In order to gather dynamic information about the indirect call sites, we executed the selected benchmark programs in the emulator of Texas Instruments' TMS470R1x C Source Debugger.

Using the conservative slicing approach, we achieved interprocedural slices where the average sizes were about 52% of all the instructions. This means that the slicing of binary executables can be an efficient tool for many applications. With the static improvements, we were able to further reduce the sizes by some 1% - 4 %. Based on our studies, the moderate improvement in the size of interprocedural slices can be mostly be attributed to the conservative handling of the memory access of the called functions and the high number of unresolved function calls.

The dynamic improvements revealed the fact that there is a high correlation between the reduction of the call edges and the size of the program slices. Where the number of indirect function calls could be determined and a big reduction of the call edges could be achieved, the computed slices were much smaller than in the conservative case.

My own contribution

The results of our studies introduced in the first part of the thesis was motivated by the fact that although there are many potential applications of the interprocedural program slicing of binary executables, there were no previous studies which satisfactorily covered this topic. We presented the interprocedural slices of binary executables with the help of dependence graphs, where the detailed description of building the control-, data- and system dependence graphs are the author's own contribution. The author's own work is an improved data flow analysis based on the lattice, while the design of this lattice is not the work of the author. The improvements of the call graph with dynamically gathered information are the joint work of the author and her co-authors, as are the design and evaluation of the experimental results. Except for the implementations of the control flow graph, all implementations of the methods presented in the thesis and experiments performed are the sole work of the author.

Static Execute After and Static Execute Before relations

Though program slicing is potentially suitable for determining the dependencies among the program components, the general solutions for program slicing are not effectively usable for large programs. The reason is twofold: firstly the program representation of a program with millions of lines of code can be extremely huge; secondly in many cases it is not necessary to determine dependencies at the same level as that for slicing.

Many applications determine the dependencies among the procedures of the given program just with the call graph [7], and the dependencies among the classes just with some cohesion metrics [8; 27]. Although these methods are quite simple, they are not safe and it is not hard to show that they do not always identify real dependencies.

In the second part of the thesis we present a technique which is not just readily computable, but it is a safe approximation of the procedure level and class level dependencies of the program. The novel technique has a high precision at the procedure level and the class level compared to the usual results obtained using conventional program slicing techniques.

II/1. Defining Static Execute After (SEA) and Static Execute Before (SEB) relations and a suitable program representation for determining these relations. Algorithms for computing the SEA and SEB relations.

Our goal was to provide an alternative way of approximating the dependencies among the procedures of the program. Our approach was motivated by Apiwattanapong [1], who introduced the notion of *Execute After* relation and applied it in dynamic impact analysis. According to the definition, the procedures f and g are in Execute After relation if and only if any part of g is executed after any part of f in any of the selected set of executions of the program.

As a static counterpart of this approach, we define the *Static Execute After* (SEA) relation. We say that $(f, g) \in \mathsf{SEA}$ if and only if it is possible that any part of g may be executed after any part of f[1]. As the notion of the backward slice is the dual of the forward slice, the *Static Execute Before* (SEB) relation can be determined as a dual counterpart of the SEA. The procedures f and g are in SEB relation if and only if it is possible that any part of g may be executed before any part of f.

According to Apiwattanapong *et al.* [1] and Beszédes *et al.* [3], the formal definition of the SEA relation is the following:

$$\mathsf{SEA} = \mathsf{CALL} \cup \mathsf{SEQ} \cup \mathsf{RET}[\cup \mathsf{ID}],$$

where

$$\begin{aligned}
(f, g) \in \mathsf{CALL} \quad &\Longleftrightarrow \quad f \text{ (transitively) calls } g,\\
(f, g) \in \mathsf{SEQ} \quad &\Longleftrightarrow \quad \exists h : f \text{ (transitively) returns into}\\
&\qquad\quad h, \text{ and after that } h \text{ (transitively)}\\
&\qquad\quad \text{calls } g\\
(f, g) \in \mathsf{RET} \quad &\Longleftrightarrow \quad f \text{ (transitively) returns into } g
\end{aligned}$$

or rather the ID is the identity relation that can optionally be a part of SEA, since a slice also contains the criterion itself and every change in a function f can affect any part of f from an impact analysis point of view.

We have to build a suitable program representation in order to determine the sets of SEA relations. With the traditional call graph representation [23] this is not sufficient,

[1]Entering a procedure and leaving a procedure are also called the events of the procedure.

because it tells us nothing about the order of the procedure calls within a procedure. On the other hand, an *Interprocedural Control Flow Graph (ICFG)* [19] contains a lot of information that is not related to procedure calls.

In our *Interprocedural Component Control Flow Graph (ICCFG)*, each procedure is represented by a *Component Control Flow Graph – (CCFG)* which contains an entry node and several component nodes. We get these component nodes by determining the strongly connected subgraph of the control flow graph of the procedure. Moreover, the components are connected by control flow edges. We can further reduce this component graph if we remove the components with no call sites and insert control flow edges among its predecessor and successor components. The CCFG graphs are connected by call edges.

We presented several alternative methods for computing the SEA and SEB relations. These methods have some extreme features. One of these methods depends on to the ICCFG and its traversals at the computation of the dependencies of a particular procedure. In the other methods we compute the dependencies of each procedure at the same time by crossing each node of the ICCFG just once.

II/2. Experimental comparison of SEA and SEB relations with relations computed by program slicing.

We used program slicing to demonstrate that the SEA and SEB relations are suitable for approximating the semantic dependencies among the program procedures. In our experiments, we compared the computed SEA and SEB relations with the relations computed by an appropriate program slicing tool, called CodeSurfer [11]. Naturally, the SEA relations were compared with the forward slices, while the SEB relations were compared with the backward slices. As we defined the SEA and SEB relations at the procedure level, for a comparison we had to extend program slicing to the procedure level. The program slice of a particular procedure is the set of procedures that contains at least one statement which is an element of one of the program slices starting from any statement of the given procedure.

For a comparison, we used the precision and recall values. The precision score is defined as the ratio of the number of procedures which are identified by either the slicer or the SEA (SEB) relation and the number of procedures identified by slicing. The recall score is defined as the extent of the properly identified dependencies based on slicing.

A program slice is computable by the appropriate traversal of the system dependence graph, starting from an arbitrary criterion [12]. The traversed edges are the control dependence, the data dependence, the parameter and the summary edges. Hence each dependency can only occur among the program points which are connected by a control flow path. This means that the procedural level program slice of a particular procedure must be a real subset of its SEA or SEB relations. So the recall score should be 100% in every case.

This assumption was not fulfilled in a comparison of the results of the forward slices got via the CodeSurfer program slicing tool and the SEA sets, while it was fulfilled in the backward cases. We elaborate on this bug in the Appendix section of the thesis. Because of this bug, we just compared the SEB sets and the relations identified by backward slicing. In our experiments, we examined 29 C programs. In most cases the precision score was around 90%. This high precision arose from the fact that the average difference between the SEA (SEB) relations and the slicing relations was some 4%. Since both the slicing and the determination of the SEA (SEB) relations are the results of an appropriate graph traversal, the graph size of the given representation is not a marginal question. To test the efficiency of our method, we identified and compared the dependence graphs of some big C++ programs. Although the differences in the sizes of these graphs are enough to show that the computation of the SEA (SEB) relations requires fewer resources, it is interesting to see that the determination of the ICCFG was straightforward, while the determination of the SDG was not possible for `mozilla`, which has over a million lines of code.

II/3. Hidden dependencies in object-oriented programs; experimental investigation of SEA and SEB relation in object-oriented programs.

Hidden dependencies are those dependencies of a program which are not explicitly readable from the source of the program. In the investigation of the hidden dependencies of the classes of object-oriented programs, our aim was to collect all potential relations among the classes. Many applications, such as change propagation or regression testing require the safe determination of the dependencies among the classes of the object-oriented programs. There are a number of ways of determining the explicit relationships in the program. Unfortunately in most cases there are many other dependencies among the classes, whose discovery is very difficult. These dependencies are called the hidden dependencies of the system.

In order to determine the dependencies among the classes, we extended the SEA and SEB relations to the class level. Of course, in its basic form the SEA and SEB algorithm has the disadvantage that it captures the data flow which is realized only through procedure calls. Data flows between global variables or direct class member variables are invisible to the algorithm due to the lack of corresponding nodes in the ICCFG. In the thesis, we present possible program transformations which could help us to discover these dependencies.

In our experiments where we investigated C++ and Java programs, we showed that the number of explicit dependencies was much lower than the number of hidden dependencies. This means that the application based only on these metrics cannot achieve a safe result during the search for dependencies. On examining the number of the dependencies connected to the individual classes, we found that the dependencies tend to form clusters. The members of a cluster have the same number of dependencies as the others have in the same cluster. The high value of the hidden dependencies and the existence of large clusters are not good because they indicate that the maintenance of the program can be difficult and expensive. Since the number of the hidden dependencies related to a particular class correlates with the size of the SEA (SEB) set of the class, the determination of the latter relations can help us to recognize the hard-to-maintain classes of the system and to find and eliminate the larger clusters.

My own contribution

The chief goal introduced in the second part of the thesis was to approximate the dependencies among the procedures and the classes of the programs we analyzed in a precise and effective way. For this reason, we introduced the SEA and SEB relations, and the ICCFG program representation. These results are the joint work of the author and her co-authors. The algorithms for the computation of the SEA and SEB relations and the related measurements were the sole results of the author. The design and the implementation of the comparison between program slicing and the SEA (SEB) method were also the sole work of the author. The design of the comparison of object-oriented metrics and the SEA (SEB) relations was a joint effort, while the implementation of the experiments related to C/C++ programs and the determination of the SEA (SEB) relations in practice were also the sole work of the author.

Conclusions

In the first part of the thesis we focused on the interprocedural program slicing of binary executables. We presented a conservative slicing method based on the traversal of dependence graphs. We improved this technique via the refinement of the dependence

	[17]	[16]	[4]	[14]	[13]
I/1	●				
I/2	●	●			
I/3	●	●			
II/1			●	●	●
II/2			●	●	●
II/3			●		

Table 1: The relation between the thesis topics and the corresponding publications.

graphs with static and dynamic information. We implemented all our solutions and evaluated the novel method on some selected examples.

In the second part of the thesis, we presented the Static Execute After and the Static Execute Before relations and we approximated the dependencies of high level languages at the procedure level and class level with the SEA and SEB relations. We introduced one appropriate program representation and several alternative methods to determine these relations. Although many of these relations are not real dependencies, this method is suitable for finding the dependencies where there is only a semantic dependence between two components.

Although Static Execute After and Static Execute Before relations contain many false dependencies, we demonstrated via experiments that these relations approximate the results of static program slicing to a high accuracy at the procedure or class level. Since the determination of these relations is not so expensive as the determination of program slicing, it can be a useful tool in applications involving program slicing. Our experimental results also revealed the fact that there is a significant correlation between the number of SEA and SEB relations and the number of direct and indirect couplings of classes in object-oriented programs. Since these relations are suitable for uncovering such hidden dependencies, which are undetectable when other simple tools are used, these relations can be used to make more reliable applications, such as those for impact analysis, change propagation and testing.

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

Acknowledgements

Both parts of the thesis contain empirical results which are the results of a team effort. My thanks goes to Ákos Kiss, Gábor Lehotai, Árpád Beszédes, Rudolf Ferenc, Tamás Gergely, Péter Siket, István Siket, Zoltán Sógor and Tibor Gyimóthy for their invaluable help and useful advice.

Judit Jász, May 8., 2009

References

[1] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 432–441, May 2005.

[2] J. Bergeron, Mourad Debbabi, M. M. Erhioui, and Béchir Ktari. Static analysis of binary code to isolate malicious behaviors. In *WETICE '99: Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, pages 184–189, Washington, DC, USA, 1999. IEEE Computer Society.

[3] Árpád Beszédes, Tamás Gergely, Szabolcs Faragó, Tibor Gyimóthy, and Ferenc Fischer. The dynamic function coupling metric and its use in software evolution. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 103–112. IEEE CS, March 21–23, 2007.

[4] Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth, Tibor Gyimóthy, and Václav Rajlich. Computation of static execute after relation with applications to software maintenance. In *Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07)*, pages 295–304. IEEE Computer Society, October 2007.

[5] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 105–113, March 2001.

[6] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[7] Shawn A. Bohner and Robert S. Arnold, editors. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[8] Lionel C. Briand, Jürgen Wüst, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 475–482, September 1999.

[9] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural static slicing of binary executables. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 188–195, October 1997.

[10] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.

[11] GrammaTech's CodeSurfer.
`http://www.grammatech.com/products/codesurfer`.

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

[13] Judit Jász. Static execute after algorithms as alternatives for impact analysis. *Peryodica Politechnica*, page Submitted paper, Budapest, 2009.

[14] Judit Jász, Árpád Beszédes, Tibor Gyimóthy, and Václav Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM'08)*, pages 137–146. IEEE Computer Society, October 2008.

[15] Daniel Kästner and Stephan Wilhelm. Generic control flow reconstruction from assembly code. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 46–55, New York, NY, USA, 2002. ACM.

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

[17] Ákos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. Interprocedural static slicing of binary executables. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 118–127, September 2003.

[18] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(2):155–163, 1988.

[19] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103. ACM Press, January 1991.

[20] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, 30(6):291–300, June 1995.

[21] 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.

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

[23] Barbara G. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

[24] Standard Performance Evaluation Corporation (SPEC). SPEC CINT2000 benchmarks.

[25] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[26] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigen, 1979.

[27] F. George Wilkie and Barbara A. Kitchenham. Coupling measures and change ripples in C++ application software. *Journal of Systems and Software*, 52(2–3):157–164, 2000.