Software Quality Journal, 13, 227–245, 2005

Using Dynamic Information in the Interprocedural Static Slicing of Binary Executables

ÁKOS KISS akiss@inf.u-szeged.hu
JUDIT JÁSZ jasy@inf.u-szeged.hu
TIBOR GYIMÓTHY gyimi@inf.u-szeged.hu
Department of Software Engineering, University of Szeged, Hungary

Abstract. Although the slicing of programs written in a high-level language has been widely studied in the literature, relatively few papers have been published on the slicing of binary executable programs. The lack of existing solutions for the latter is really hard to understand since the application domain for slicing binaries is similar to that for slicing high-level languages. Furthermore, there are special applications of the slicing of programs without source code like source code recovery, code transformation and the detection of security critical code fragments. In this paper, in addition to describing the method of interprocedural static slicing of binaries, we discuss how the set of the possible targets of indirect call sites can be reduced by dynamically gathered information. Our evaluation of the slicing method shows that, if indirect function calls are extensively used, both the number of edges in the call graph and the size of the slices can be significantly reduced.

Keywords: static slicing, interprocedural slicing, binary executables, call graph, indirect function call, dynamic information

1. Introduction

Program slicing is a technique originally introduced by Weiser (1984) for automatically decomposing a program by analysing its control and data flow. Since the introduction of the original concept of slicing, various notions of static and dynamic program slices have been proposed (Horwitz et al., 1990; Harman et al., 2003; Krinke, 2003; Canfora et al., 1998; Korel and Laski, 1988; Beszédes et al., 2001). Good surveys of slicing methods can be found in Tip (1995) and Binkley and Gallagher (1996). The *program dependence graph*-based *static* slicing algorithm of Ottenstein and Ottenstein (1984) for single-procedure programs was extended by Horwitz etal. (1990) to slicing multi-procedure programs using the notion of a *system dependence graph*.

These algorithms were originally developed for slicing high-level structured programs and so 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 (Agrawal, 1994; Ball and Horwitz, 1993; Choi and Ferrante, 1994; Shapiro and Horwitz, 1997; Antoniol et al., 1999; Kumar and Horwitz, 2002).

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 (1997) presented a technique for the intraprocedural slicing of binary executables, but we are not aware of any usable interprocedural solution. Bergeron et al. (1999) suggested using dependence graph-based interprocedural slicing to analyse binaries but they did not discuss the handling of the problems that 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 that 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 special applications aid code understanding, source code recovery, bugfixing and code transformation.) Security is also becoming an increasingly important topic. The detection of malicious code fragments, especially in COTS components, is now a major concern of researchers. The slicing of binary executables can be a useful method for helping extract security critical code fragments (Bergeron et al., 1999).

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

Static slicing is a useful analysis method for maintenance and program understanding because, in this way, the irrelevant parts of the program can be 'sliced away'. However, static slices of even small programs are in many cases too large (Beszédes et al., 2002), which means that often too large portions of the code still need to be examined. In our previous paper (Kiss et al., 2003) we applied interprocedural static slicing on real life binary executables and achieved an average slice size of 56–68%. We looked for ways of improving static data dependence analysis but found that, although the number of data dependences could be reduced significantly, sometimes even by 59%, this reduction was not reflected in the size of the slices, which only dropped by 1–4%.

Since the static slices turn out to be so large, alternatives are needed to compute slices that are useful in practice. This was also the motivation for union slicing introduced in Beszédes et al. (2002). The concept described therein was to obtain a more precise slice by computing the union of dynamic slices, which is a good approximation of the real dependences if enough test cases are used. The drawback of this approach is that it is very expensive as it is based on the computation of dynamic slices. The computation of the complete trace makes the execution of programs slower by orders of magnitude and, moreover, in the case of real life programs the trace size can be huge.

Mock et al. (2002) suggested using dynamic points-to data to reduce the size of static slices of C programs. They made the static call graph more precise by using dynamic information and applied the so-reduced call graph during the slice computation. Their results show that, in case of C programs which intensively use indirect function calls, the size of slices can be markedly reduced. Naturally, a slice computed this way is usually unsafe, i.e. there may be dependences in the sliced program which are not present in the slice. Nevertheless, a big advantage of this approach is that the reduced slices can be computed with little effort. Moreover, there are applications of slicing where the 'safe' property of slices is not critical. For example debugging could be started with a reduced slice and only if the problem cannot be identified with the help of it is the usually substantially larger static slice to be used.

Since the call graph of binary executables is usually quite big owing to the indirect function calls, we chose the approach of Mock's to improve static slicing. An additional motivation was that the static points-to analyses, developed for high-level programming languages, are hard to apply to binary code.

In this paper, in addition to describing the method for the interprocedural static slicing of binaries, we explain how the set of the possible targets of indirect call sites can be reduced by dynamically gathered information. Although with the use of dynamic information slices may become unsafe, with a sufficient number of test cases we can provide a good approximation of the static call graph. Our evaluation of this method shows that if indirect function calls are extensively used and an aggressive edge elimination approach is applied both the number of edges in the call graph and the size of the slices can be substantially reduced.

The rest of this paper is organized as follows. In Section 2 we discuss the problems that arise during the control flow analysis of binary executables and propose solutions for them. In Section 3 and its subsections we present our solution for the interprocedural slicing of binary executables. In Section 4 we discuss our previous work on statically improving the precision of the slicing of binaries and offer a new method for reducing the number of the possible targets of an indirect function call by means of dynamically gathered information. In the section following we present our experimental results. In Section 6 we give an overview of related works then, in the last section, we provide a short summary and ideas for future research.

2. Control flow analysis of binary executables

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. However, the control flow analysis of a binary executable has a number of problems associated with it 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 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

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

offset of one byte can and will yield completely false results. 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. Figure 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* are determined, which will constitute the nodes of the CFG, then the blocks are further grouped to represent *functions*. Additionally, 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 with *control flow*, *call* and *return edges* to represent the appropriate possible control transfers during the execution of the program. This requires the behaviour analysis of machine instructions. Even the high number of the types of instructions may be hard to cope with, but the hardest problem is rised by control transfer instructions, where the *target cannot be determined unambiguously*. In high-level languages only indirect function calls fall in this category, but on binary level intraprocedural control transfer may be represented in 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, 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 linked to all the possible targets of the indirect control transfers. Figure 3 shows a function containing an indirect call and two other functions as possible targets of the call. The corresponding CFG is shown as well.

Another type of problems is when control is transfered between functions in a way different from function calls. *Overlapping* (or multiple entry) and *cross-jumping* functions, which usually do not occur in high-level languages and result from aggressive interprocedural compiler optimizations, are typical examples of this problem. Since, in the case of these constructs, the exit node of the control transfering function is not reached, a control flow edge has to be inserted between the exit nodes of the functions to make up for it.

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
```

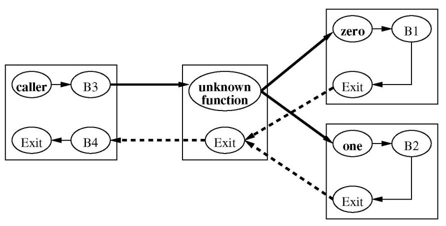


Figure 3. An indirect function call with two possible targets in ARM. In the CFG thin solid vectors represent control flow edges, thick solid vectors are call edges, while thick dashed lines are return edges. Basic block nodes represent the corresponding code fragments, as denoted on the left.

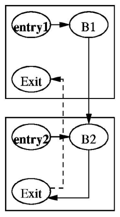
Figure 4 gives an example for the overlapping functions and shows the CFG representation of them.

During the discussion of the problems above we left some questions open: 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? It is not possible to furnish a nice 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 (TIS Committee, 1995; Microsoft Corporation, 1999) can store extra information along with the raw binary data. 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. Similarily, relocation information can be most helpful in determining the targets of indirect function calls and ambiguous transfer controls. Usually hand-written assembly code can also be analysed with no, or very little, extra user input.

Code | **CFG** | **CDG**

```
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
```



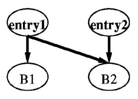


Figure 4. Two overlapping functions in ARM. In the CFG thin solid vectors are control flow edges while thin dashed vectors represent compensation control edges. In the CDG thick solid vectors denote control dependence. Basic block nodes represent the corresponding code fragments, as denoted on the left.

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. 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, the necessary information can be retrieved relatively easily.

3. The dependence graph-based slicing of binary executables

To slice a binary executable we perform the following steps: first, we build an interprocedural control flow graph as outlined in Section 2, then we perform a control and data dependence analysis for each function found in the CFG. These result in a control dependence graph (CDG) and a data dependence graph (DDG) for each function, which together form a program dependence graph (PDG). These PDGs can then be used to compute intraprocedural slices or, by incorporating them in a system dependence graph (SDG), interprocedural slices can be computed (Horwitz et al., 1990).

The following subsections describe each step mentioned above in more detail and point out the issues we are confronted with when dealing with binary executable programs.

3.1. Building the PDG

One component of the PDG of a function is the CDG, which represents control dependences between 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 use the algorithm described in Lengauer and Tarjan (1979) to find post-dominators and then we build the actual CDG according to Ferrante et al. (1987). The resulting graph consists of nodes representing basic blocks and function entries, and *control dependence edges* connecting these nodes.

One feature peculiar to binary programs is that instructions may belong to multiple functions due to overlapping and cross-jumping, a situation that never occurs in high-level structured programming languages. This leads to instructions that may depend on multiple function entry nodes. The third column of Figure 4 shows the CDG of two overlapping functions.

The other part of a PDG is the 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 what 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

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

Figure 5. Computing U_f and D_f

determine only 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 4.1.

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 those 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 that in high-level programs, in binaries the parameter list of procedures is not defined explicitly 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 Horwitz et al. (1990)) representing the used and defined arguments of every instruction in function f itself and in functions called (transitively) from f, as given in Figure 5. 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.

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, which, of course, each depend on their 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 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 those 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, 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 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.

In Figure 6 we provide the Thumb assembly listing and the CFG of the classic example program (which computes the sum and product of the first N natural numbers). In Figure 7 the PDG of one of the functions is shown.

3.2. *Interprocedural slicing using the SDG*

To compute interprocedural slices the individual PDGs of functions 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 Reps et al. (1994). The resulting graph is the system dependence graph (SDG) of the program.

Similar to Section 3.1, call edges connect basic block and function entry nodes, but the real dependences come from the call instructions. To avoid missing dependences, the SDG needs to be augmented with new dependence edges. The SDG built this way can be used to compute slices using the two-pass algorithm of Horwitz et al. (1990) w.r.t. sets of argument nodes.

Figure 8 presents a small portion of the SDG of the example program containing a call site and the entry point of the corresponding called function.

4. Improving the slicing of binary executables

Although the program and system dependence graphs built as described in Section 3 are safe, they are overly conservative. Both control and data flow information can be made more precise either by refining static analyses or with the aid of dynamically obtained information.

Code

```
add:   add r0, r1, r0     ;B1        main:  push {r4,r5,lr}    ;B7
       mov pc, lr                           add sp, #-8
                                            bl readin
mul:   push {r4,r5,lr}    ;B2               add r5, r0, #0     ;B8
       add r4, r1, #0                       mov r0, #0
       add r5, r0, #0                       str r0, [sp, #0]
       mov r3, #0                           mov r0, #1
       mov r2, #1                           str r0, [sp, #4]
       cmp r2, r4                           mov r4, #1
       bgt .l2                              cmp r4, r5
.l1:   add r0, r3, #0     ;B3               bge .l4
       add r1, r5, #0                .l3:   ldr r0, [sp, #0]   ;B9
       bl add                               add r1, r4, #0
       add r3, r0, #0     ;B4               bl add
       add r0, r2, #0                       str r0, [sp, #0]   ;B10
       mov r1, #1                           ldr r0, [sp, #4]
       bl add                               bl mul
       add r2, r0, #0     ;B5               str r0, [sp, #4]   ;B11
       cmp r2, r4                           add r4, #1
       ble .l1                              cmp r4, r5
.l2:   add r0, r3, #0     ;B6               blt .l3
       pop {r4,r5,pc}                .l4:   ldr r0, [sp, #0]   ;B12
                                            ldr r1, [sp, #4]
                                            bl writeout
                                            add sp, #8         ;B13
                                            pop {r4,r5,pc}
```

CFG

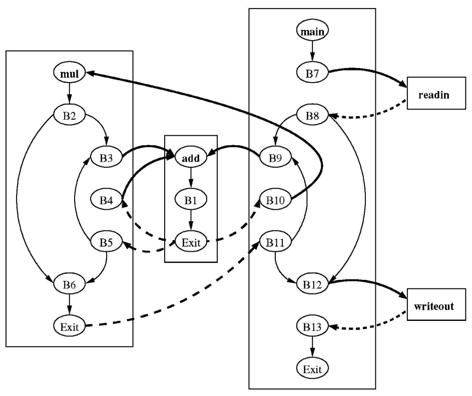


Figure 6. A Thumb program for computing the sum and product of the first N natural numbers. In the CFG thin solid vectors represent control flow edges, thick solid vectors are call edges, while thick dashed lines are return edges. Basic block nodes represent the corresponding code fragments, as denoted in the listing.

4.1. *Refining static analyses*

In previous work (Kiss et al., 2003) we furnished two static methods for improving the slicing of binaries. The first approach described how architecture specific knowledge could be used to extract information from function prologs and epilogs in order to determine the output parameters of functions more precisely.

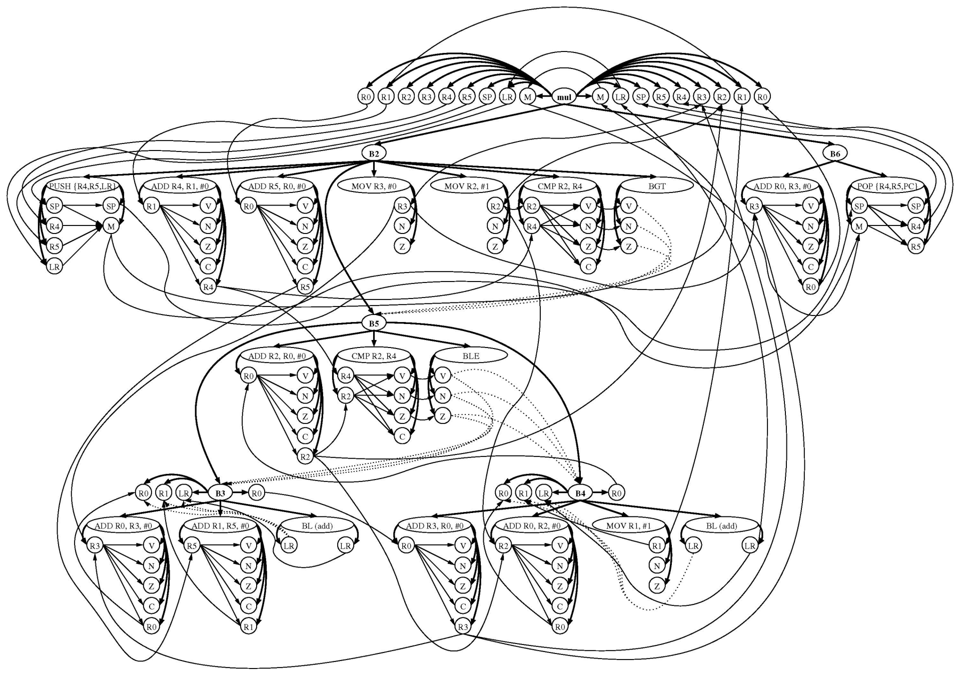


Figure 7. The PDG of function `mul` of the example program presented in Figure 6. Thick and thin solid vectors represent control and data dependence edges, respectively, while dotted vectors are compensation control dependence edges. The Rx, LR and SP arguments represent registers, V, N, Z, and C are flags and M stands for the memory.

The second problem we addressed was a limitation of binary executables, namely that at the binary level the high-level concept of function parameters and local variables generally do not exist. Such constructs are usually mapped to registers and to a special portion of the memory called the stack. Since the memory model outlined in Section 3.1 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 proposed an improved memory model (a modified data dependence analysis and a propagation algorithm) to aid the analysis.

In our experiments we found that although the number of data dependence and summary edges in the SDGs fell by 27% and 32% on average, and in some cases even by 59% and 36%, the size of the slices dropped only by 1–4%. Our investigations revealed that the high number of unresolved indirect function calls is the key reason why the slices were too large and the improvements were only moderate.

Antoniol et al. (1999) experimented with static points-to analysis of C programs, focusing on function pointers and their effect on the call graph. Although their results are 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.

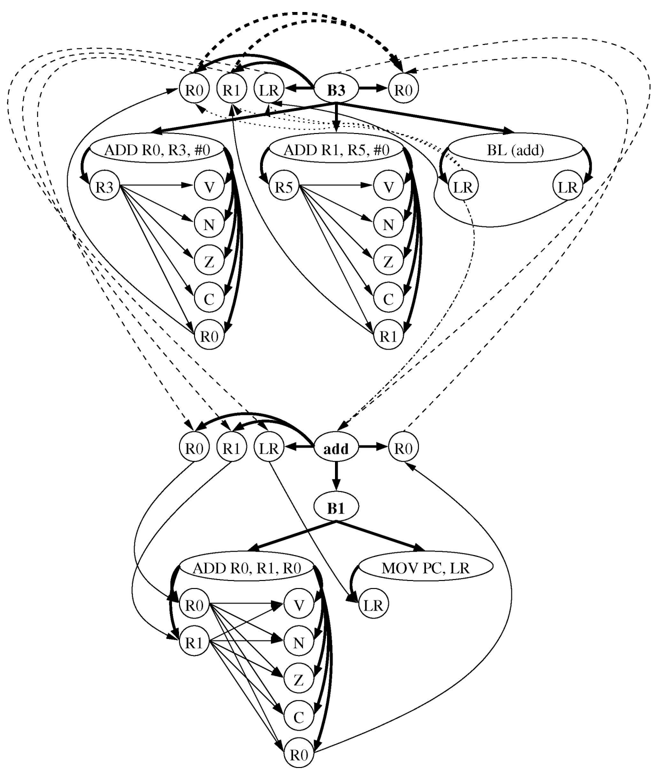


Figure 8. A portion of the SDG of the example program given in Figure 6, showing the function `add` and call site B3 in the function `mul`. Thick and thin solid vectors represent control and data dependence edges, respectively, thick dashed vectors are summary edges, thin ones denote parameter and call edges, while dotted and dot-dashed vectors are for control and call compensation edges.

4.2. Improving the call graph with dynamic information

Mock et al. (2002) 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, and less to data pointers. 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 a function pointer and call via a function pointer are a 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 enable 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, and write each address to the disk. 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.

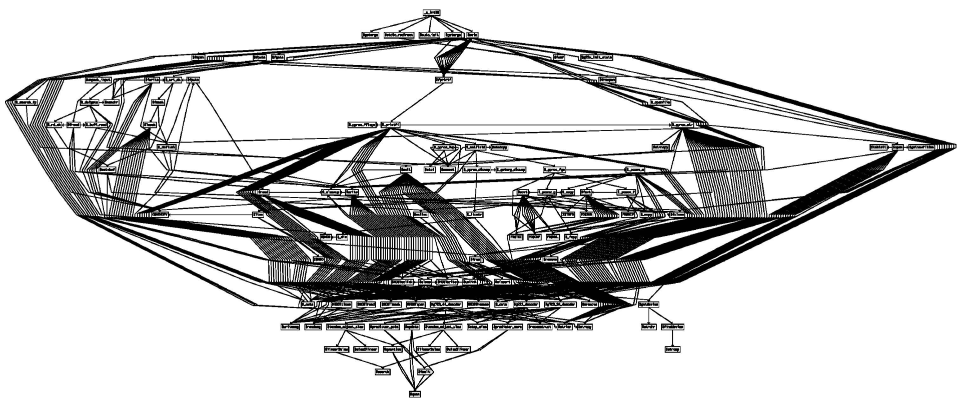


Figure 9. The statically computed call graph of the program `decode`.

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 realized 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. Those call sites which were not executed during any invocation of the application have no dynamic information associated with them so 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 5 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 shows the call graph part of an example program named `decode`, while Figure 10 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 just the summary edges in the SDG need be recomputed and then the interprocedural static slice can be computed in the usual way.

5. Experimental results

We implemented a slicer for statically linked binary ARM executables and evaluated it on programs taken from the SPEC CINT2000 (Standard Performance Evaluation Corporation

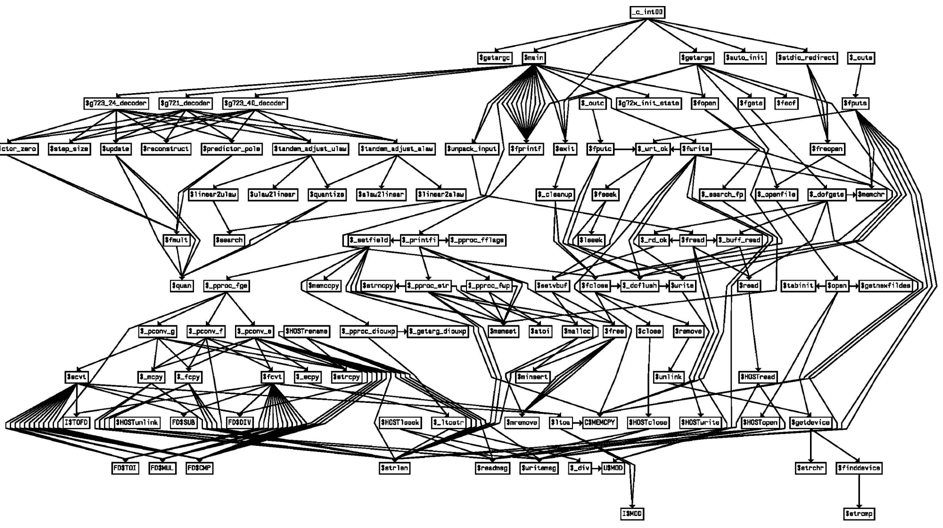


Figure 10. The statically computed call graph shown in Figure 9 made more precise with the help of dynamically gathered information.

(SPEC), 2000) and MediaBench benchmark suites (Lee et al., 1997). 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 1 we also state the number of the 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.

In Table 2 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). Only those functions are considered to be potential targets of indirect calls of which the address is explicitly referenced since an indirect function call requires the address of the called function. In the implementation of the slicer these functions are identified with the help of extra information stored in the executable files.

Table 1. Size of benchmark programs.

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 2. Indirect function calls.

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)

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. We used the test inputs that came with the benchmark suites to achieve as good a code coverage as possible. In Table 3 we list for each program the number of functions present in the binary code, the number of functions reachable from the entry point of the program according to the statically computed call graph, and the number of functions called during the executions of the program. 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. The number of executed functions tells us 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 get a better ratio, since a big portion of the code turns out to be never executed, even though static analysis marks it reachable. This is usually the case when the number of indirectly called functions is high. After all, unless all realizable paths of execution can be covered by the test inputs the call graph resulting 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 collected dynamic information we can make the call graph at the indirect call sites and their targets more accurate. Table 4 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 4.2). Here the number of indirect call edges is

Table 3. Code coverage of inputs.

Program	All functions	Reachable functions	Executed functions
ansi2knr	5 (+114)	5 (+97)	4 (+57)
decode	26 (+109)	21 (+91)	21 (+45)
bzip2	73 (+119)	51 (+102)	38 (+53)
toast	86 (+124)	72 (+107)	60 (+54)
sed	102 (+142)	92 (+128)	57 (+70)
cjpeg	381 (+149)	327 (+133)	134 (+62)
osdemo	1186 (+145)	675 (+127)	122 (+85)

Table 4. Call edges.

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)

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, where 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. To avoid bias from applying a given selection strategy we decided to compute slices for each instruction of those functions that was compiled from the sources (not added during the linking process) and called during the executions of the benchmark programs. In Table 5 we list the average number of instructions in the 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% compared to the static approach. Two programs using indirect function calls only rarely (`decode` and `toast`) had 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.

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

Table 5. Average number of instructions in slices (no criteria from library code).

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 6. Average number of instructions in slices (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	a9631	8959	8947	8344
sed	14431	11307	11292	10981
cjpeg	14640	28911	28870	11127
osdemo	24319	97890	97850	48514

reason in Table 6 we list results on slices computed for those functions which originate from library code as well. As these figures are similar to those shown in Table 5, we may draw a similar conclusion as before. With those 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.

6. Related work

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 (Debray et al., 2000), 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 use an intraprocedural static slicing technique in their binary executable editing library called EEL (Larus and Schnarr, 1995). They utilise slicing to improve the precision of control flow analysis in the case of indirect jumps mainly occurring in the compiled form of case statements. With the help of backward slicing they are able to analyse such constructs in an architecture and compiler-independent way.

Cifuentes and Fraboulet also make use of intraprocedural slicing for solving indirect jumps and function calls in their binary translation framework (Cifuentes and Fraboulet, 1997). Bergeron et al. suggest using interprocedural static slicing for analysing binary code to detect malicious behavior (Bergeron et al., 1999). The computed slices should be verified against behavioral specifications in order to statically detect potentially malicious code. However they did not elaborate on the potential problems of analysis binary executables nor did they present any experimental results.

Antoniol et al. examined static points-to analysis of C programs and investigated the impact of function pointers on the call graph (Antoniol et al., 1999). Mock et al. analysed the feasibility of improving static slicing with dynamically gathered points-to data (Mock et al., 2002). 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.

7. Conclusions and future work

In this paper we described how interprocedural static slicing can be applied to binary executables and we evaluated our new method on real life applications. We experimented with two static improvements but found that the size of the slices did not decrease significantly. We described how superfluous edges can be removed from the statically computed call graph with the help of dynamically gathered information. The experiments with this method demonstrated that the slice size could be dramatically reduced if the application being analysed 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.

Although previous work shows that data dependences are less important than the call graph, improving the data dependence analysis of binaries is still worth investigating since the memory model of low-level and high-level programs differ significantly. In the future we plan to improve our method of analysis for binary executables.

References

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

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

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

Bergeron, J., Debbabi, M., Erhioui, M.M., and Ktari, B. 1999. Static analysis of binary code to isolate malicious behaviors, In *Proc. IEEE International Workshop on Enterprise Security*.

Beszédes, Á., Faragó, C., Szabó, Z.M., Csirik, J., and Gyimóthy, T. 2002, Union slices for program maintenance. In *Proc. IEEE International Conference on Software Maintenance (ICSM 2002)* pp. 12–21.

Beszédes, Á., Gergely, T., Szabó, Z.M., Csirik, J., and Gyimóthy, T. 2001. Dynamic slicing method for maintenance of large C programs, In *Proc. Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pp. 105–113.

Binkley, D. and Gallagher, K.B. 1996. Program slicing, *Advances in Computers* 43: 1–50.

Canfora, G., Cimitile, A., and De Lucia, A. 1998. Conditioned program slicing, In *Information and Software Technology Special Issue on Program Slicing*, Vol. 40, pp. 595–607.

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

Cifuentes, C. and Fraboulet, A. 1997. Intraprocedural static slicing of binary executables, In *Proc. International Conference on Software Maintenance*, pp. 188–195.

Debray, S.K., Evans, W., Muth, R., and Sutter, B.D. 2000. Compiler techniques for code compaction, *ACM Trans. Program. Lang. Syst.* 22(2): 378–415.

Ferrante, J., Ottenstein, K.J., and Warren, J.D. 1987. The program dependence graph and its use in optimization, *ACM Trans. Program. Lang. Syst.* 9(3): 319–349.

Harman, M., Binkley, D.W. , and Danicic, S. 2003. Amorphous program slicing, *Journal of Systems and Software* 68(1): 45–64.

Horwitz, S., Reps, T., and Binkley, D. 1990. Interprocedural slicing using dependence graphs, *ACM Trans. Program. Lang. Syst.* 12(1): 26–61.

Kiss, Á., Jász, J., Lehotai, G., and Gyimóthy, T. 2003. Interprocedural static slicing of binary executables, in *Proc. Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pp. 118–127.

Korel, B. and Laski, J. 1988. Dynamic program slicing, *Information Processing Letters* 29(2): 155–163.

Krinke, J. 2003. Advanced slicing of sequential and concurrent programs, Ph.D. thesis, Universität Passau.

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

Larus, J.R. and Schnarr, E. 1995. EEL: Machine-independent executable editing, *ACM SIGPLAN Notices* 30(6): 291–300.

Lee, C., Potkonjak, M., and Mangione-Smith, W.H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons Systems, In *Proc. International Symposium on Microarchitecture*, pp. 330–335.

Lengauer, T. and Tarjan, R.E. 1979. A fast algorithm for finding dominators in a flowgraph, *ACM Trans. Program. Lang. Syst.* 1(1): 121–141.

Microsoft Corporation. 1999. Microsoft portable executable and common object file format specification version 6.0. `http://www.microsoft.com/hwdev/hardware/PECOFF.asp`.

Mock, M., Atkinson, D.C., Chambers, C., and Eggers, S.J. 2002. Improving program slicing with dynamic points-to data, In *Proc. 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 71–80.

Ottenstein, K.J. and Ottenstein, L.M. 1984. The program dependence graph in a software development environment, In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 177–184.

Reps, T., Horwitz, S., Sagiv, M., and Rosay, G. 1994. Speeding up slicing, In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 11–20.

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

Standard Performance Evaluation Corporation (SPEC). 2000. SPEC CINT2000 Benchmarks. `http://www.spec.org/osg/cpu2000/CINT2000/`.

Tip, F. 1995. A survey of program slicing techniques, *Journal of Programming Languages* 3, 121–189.

TIS Committee. 1995. Tool interface standard (TIS) executable and linking format (ELF) version 1.2. `http://www.x86.org/ftp/manuals/tools/elf.pdf`.

Weiser, M. 1984. Program slicing, *IEEE Trans. Software Eng.* 10(4): 352–357.

Ákos Kiss obtained his M.Sc. in Computer Science from the University of Szeged in 2000. He is currently working on his Ph.D. thesis and his chosen field of research is the analysis and optimization of binary executables. He was the chief programmer of a code compaction project which sought to reduce ARM binaries. He is also interested in GCC and in open source development.

Judit Jász obtained her M.Sc. in Computer Science recently from the University of Szeged and is currently a Ph.D student. Her main research interest is adapting slicing methods—originally intended for high-level languages—to binary executables. She is also actively working on improving the GCC compiler.

Tibor Gyimóthy is the head of the Software Engineering Department at the University of Szeged in Hungary. His research interests include program comprehension, slicing, reverse engineering and compiler optimization. He has published over 60 papers in these areas and was the leader of several software engineering R&D projects. He is the Program Co-Chair of the 21th International Conference on Software Maintenance, which will be held in Budapest, Hungary in 2005.