Interprocedural Static Slicing of Binary Executables

Ákos Kiss, Judit Jász, Gábor Lehotai and Tibor Gyimóthy

Research Group on Artifical Intelligence
University of Szeged and Hungarian Academy of Sciences
Aradi vértanúk tere 1., H-6720 Szeged, Hungary
E-mail: {akiss,jasy,leg,gyimi}@rgai.inf.u-szeged.hu

Abstract

Although the slicing of programs written in a high-level language has been widely studied in the literature, very little work has been published on 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. In this paper we present a method for the interprocedural static slicing of binary executables. We applied our slicing method to real size binaries and achieved an interprocedural slice size of between 56%-68%. We used conservative approaches to handle unresolved function calls and branching instructions. Our current implementation contains an imprecise (but safe) memory dependence model as well. However, this conservative slicing method might still be useful in analysing large binary programs. In the paper we suggest some improvements to eliminate useless edges from dependence graphs as well.

1. Introduction

Program slicing is a technique originally introduced by Weiser in [24] for automatically decomposing programs by analysing its control and data flow. Since the introduction of the original concept of slicing various notions of program slices have been proposed: *forward* and *backward*, *static* and *dynamic* slices have been defined in [5, 6, 12, 22].

Many algorithms are available in the literature for computing static slices: the original *dataflow equation* based solution of Weiser, the *information-flow* based solution of Bergeretti and Carré in [3] and the *program dependence graph*-based approach of Ottenstein and Ottenstein in [18]. The original program dependence-graph based slicing algorithm for single-procedure programs was extended by Horwitz, Reps and Binkley in [11] for slicing multi-procedure programs using the notion of a *system dependence graph*.

The above algorithms were originally developed for slicing high-level structured programs and so do not handle unstructured control flow correctly and yield imprecise results. Another source of imprecision is the complexity of static resolution of pointers. Several modifications and corrections have been published to overcome imprecise behaviour [1, 2, 7, 13, 20].

Although the slicing of programs written in a high-level language has been widely studied in the literature, very little work has been published on the slicing of binary executable programs. Cifuentes and Frabuolet presented a technique for the intraprocedural slicing of binary executables in [8], but we are not aware of any usable interprocedural solution. Bergeron et al. in [4] suggested to use dependence graph-based interprocedural slicing to analyse binaries but they did not discuss the handling of the arising problems and neither gave any experimental result.

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 – e.g. assembly programs, legacy software and viruses: code understanding, source code recovery, bugfixing and code transformation.

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.

The main contribution of this paper is to give a method for the interprocedural static slicing of binary executables. We applied our slicing method to real size binaries and achieved an interprocedural slice size of between 56%-68%. We used conservative approaches to handle unresolved function calls and branching instructions. Here, the current implementation contains an imprecise (but safe) memory dependence model as well. However, this conservative slicing method might still be useful in analysing large binary programs. If we could produce a more precise con-

Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)
0-7695-2005-7/03 $ 17.00 © 2003 IEEE

trol flow graph and memory dependence model the size of slices might be significantly reduced. In the paper we suggest some improvements to eliminate useless edges from dependence graphs.

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 interprocedurally slice binary executables. In Section 4 we present our experimental results. In Section 5 we provide an overview of related works and, finally in Section 6, we give a summary and suggestions 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 depend on it. Although building a CFG for a program written in a high-level structured programming language like C or Pascal is usually a simple task and requires only syntactical analysis, the control flow analysis of a binary executable has a number of problems associated with it.

2.1. 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 program itself has to be recovered from its binary form. This requires that the boundaries of the low-level instructions of which the program is constructed be detected, which can again cause several problems. On architectures with variable length instructions the boundaries may not be detected unambiguously. On architectures with multiple instruction sets it may be difficult to determine the instruction set used. If the binary representation mixes code and data their separation may be also difficult.

Even if we are able to find the boundaries of the instructions there are still more hurdles to cross. To be able to analyse the control flow among the detected instructions their behaviour must be analysed, which requires much more effort than simply analysing the source code. Since the types of instructions at the binary level are much more numerous than the types of control structures at the source level. In addition, the analysis of certain kinds of instructions may present problems if it cannot be unambiguously determined where the analysed instructions transfer control. Such ambiguities do not occur in high-level structured programming languages and require special handling in our case.

After the analysis of instructions and data has been performed the boundaries of functions have to be found. Then function call sites have to be detected and the targets of the function calls need to be determined. The detection of function boundaries is not an easy task in general, but indirect function calls, where the target of the call cannot be determined unambiguously, and overlapping and cross-jumping functions (where the control flow can cross function boundaries) present further problems. Although indirect function calls also occur in high-level languages, the potential targets of indirect calls are much more numerous than in their high-level counterpart. The issue of overlapping and cross-jumping functions is also a problem that usually does not arise in high-level programs.

As is evident from the list of problems described above, the control flow analysis of binary executable programs has to overcome several difficulties. It is very hard to furnish a general solution that handles all the problems, but with more information and some architecture specific heuristics the problems become manageable.

One source of the information required to assist the control flow analysis is usually in the file that contains the binary image of the program, since most file formats (see [17, 23]) can store extra information as well as raw binary data. Compilers, assemblers and linkers usually store symbolic and relocation information in the generated files. Symbolic information may be used to separate code and data in the binary image of the program or help in detecting function boundaries and instruction set switches. Relocation information can be 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 few extra user input.

Needless to say, the kinds of information stored in the files are highly dependent on the hardware and operating system the binary executable is going to run on, the tool chain the program is generated with, and the file format used. Hence we cannot say in general how useful data can be extracted from symbolic and relocation information. Similarily, there is no standard way of analysing the behaviour of the instructions running on different platforms. Altough these issues seem to represent new problems, experience shows that they can be solved using an appropriate compiler, file format and architecture specification.

2.2. Building the CFG

Once we are able to overcome the problems presented by binary executables, we have to build the graph. First, *basic block leader* information has to be collected by analysing the instructions. The instructions following branching or function calling instructions, instructions targeted by branching instructions, first instructions of functions and instructions following instruction set switches are called leaders. Instructions between leaders form the *basic blocks* of

the program, and these blocks are further grouped to represent functions. Then the nodes in the CFG representing functions contain further nodes representing basic blocks, which again contain nodes representing instructions. A special node called the *exit node* is also added to each function node to represent the single exit points of the corresponding functions.

Control flow edges connect basic blocks in the CFG to represent the possible control flow in the program. From basic blocks ending in an instruction representing a *return from a function*, control flow edges lead to the exit node. A basic block that ends in an instruction implementing a function call is called a *call site*, while the basic block beginning with the instruction following it in the raw binary representation is called the corresponding *return site*. *Function call edges* connect the call sites with nodes representing the called functions while *return edges* connect the exit nodes of the functions with the corresponding return sites.

Special care has to be taken with instructions which branch to targets or call functions that cannot be determined. Such constructs typically arise in compiling C switch structures or function pointer uses. To handle such contructs the CFG has to be extended using special nodes. A node called the *unknown function node* is used to represent the target of the unresolved function calls. Function call edges connect this special node with all functions that can be targets of unresolved function call instructions. Also a special node called the *unknown block node* has to be added to each function that contains unresolved branching instructions to represent the target of those branches. As with the handling of unknown function node, control flow edges connect the special unknown block node with all the possible targets of the unresolved branches (which can be detected using relocation information, see Section 2.1).

Overlapping and cross-jumping functions also require special handling. If a function transfers control to another function in a way other than the function call then the exit node of the invoked function has to be connected with the exit node of the invoking function to compensate for the lack of a return edge.

2.3. Example

In Fig. 1 we list a part of a disassembled code of a program compiled for the ARM architecture with basic block boundaries already determined, while in Fig. 2 we show the corresponding CFG.

```
                    add:
00002ECC    1808    ADD R0, R1, R0              B1
00002ECE    46F7    MOV PC, LR

                    mul:
00002ED0    B530    PUSH {R4,R5,LR}             B2
00002ED2    1C0C    ADD R4, R1, #0
00002ED4    1C05    ADD R5, R0, #0
00002ED6    2300    MOV R3, #0
00002ED8    2201    MOV R2, #1
00002EDA    42A2    CMP R2, R4
00002EDC    DC0B    BGT 0x00002EF6
00002EDE    1C18    ADD R0, R3, #0              B3
00002EE0    1C29    ADD R1, R5, #0
00002EE2    F7FFFFF3  BL 0x00002ECC (add)
00002EE6    1C03    ADD R3, R0, #0              B4
00002EE8    1C10    ADD R0, R2, #0
00002EEA    2101    MOV R1, #1
00002EEC    F7FFFFEE  BL 0x00002ECC (add)
00002EF0    1C02    ADD R2, R0, #0              B5
00002EF2    42A2    CMP R2, R4
00002EF4    DDF3    BLE 0x00002EDE
00002EF6    1C18    ADD R0, R3, #0              B6
00002EF8    BD30    POP {R4,R5,PC}

                    main:
00002EFA    B530    PUSH {R4,R5,LR}             B7
00002EFC    B082    ADD SP, #-8
00002EFE    F7FFFFD5  BL 0x00002EAC (readin)
00002F02    1C05    ADD R5, R0, #0              B8
00002F04    2000    MOV R0, #0
00002F06    9000    STR R0, [SP, #0]
00002F08    2001    MOV R0, #1
00002F0A    9001    STR R0, [SP, #4]
00002F0C    2401    MOV R4, #1
00002F0E    42AC    CMP R4, R5
00002F10    DA0B    BGE 0x00002F2A
00002F12    9800    LDR R0, [SP, #0]            B9
00002F14    1C21    ADD R1, R4, #0
00002F16    F7FFFFD9  BL 0x00002ECC (add)
00002F1A    9000    STR R0, [SP, #0]            B10
00002F1C    9801    LDR R0, [SP, #4]
00002F1E    F7FFFFD7  BL 0x00002ED0 (mul)
00002F22    9001    STR R0, [SP, #4]            B11
00002F24    3401    ADD R4, #1
00002F26    42AC    CMP R4, R5
00002F28    DBF3    BLT 0x00002F12
00002F2A    9800    LDR R0, [SP, #0]            B12
00002F2C    9901    LDR R1, [SP, #4]
00002F2E    F7FFFFC6  BL 0x00002EBE (writeout)
00002F32    B002    ADD SP, #8                  B13
00002F34    BD30    POP {R4,R5,PC}
```

Figure 1. Excerpt from a disassembled binary code. Basic blocks are separated by horizontal lines. The program computes the sum and product of the numbers ranging from 1 to a read number, and writes the results.

3. The dependence graph-based slicing of binary executables

For slicing a binary executable we perform the following steps: first, we build an interprocedural control flow graph as described 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.

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

3.1. Building 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 for arbitrary control flow is defined in terms of post-dominance we use the algo-

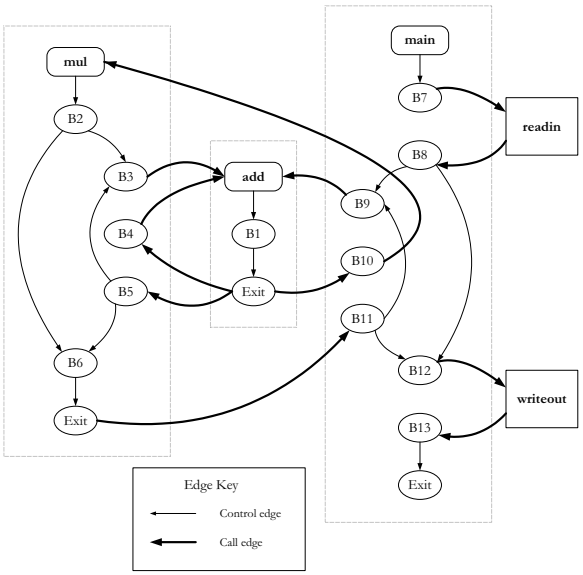

Figure 2. Control flow graph of the program given in Fig. 1.

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

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

Figure 3. Computing U_f and D_f

rithm described in [16] by Lengauer and Tarjan to find post-dominators and then we build the actual CDG according to [10]. The resulting graph consist of nodes representing basic blocks and function entries and *control dependence edges* connecting these nodes.

As a special feature of binary programs, instructions may belong to multiple functions due to overlapping and cross-jumping, a situation that never occurs in high-level structured programming languages. This causes that instructions may depend on multiple function entry nodes.

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/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 every instruction in the program and determine what registers and flags it reads or writes. The analysis does not take the register into account, which controls the flow of the program (usually called the instruction pointer or program counter), since the effect of this register is captured by the CFG and CDG. We also analyse the memory access of the instructions but only to a limited extent. We only determine whether an instruction reads or writes memory, thus representing the whole memory as only one argument. This limitation is used to circumvent the complexities of static pointer resolution. An improved solution is described in Section 3.2 and some of our ideas are stated in Section 6. The analysis results in the sets u_j and d_j for each instruction j, which contain all used and defined arguments of j, respectively. During the analysis we also determine the sets $u_j^{(a)}$ for every $a \in d_j$, which contain the arguments of j actually used to compute the value of a. Obviously $u_j = \bigcup_{a \in d_j} u_j^{(a)}$ for each instruction j, but instructions may exist where $u_j^{(a)} \subset u_j$ for a defined argument a. High-level programming languages may also have such statements, but usually they can be divided into subexpressions with only one defined argument when analysed, 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), respectively [11]) representing the used and defined arguments of all instructions in function f itself and in functions called (transitively) from f, as given in Fig. 3. 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.

Fig. 4 shows the evaluation of the U_f and D_f sets for the functions of the example program. The iteration for functions `readin` and `writeout` is not detailed but the fix-point is given.

Using the results of these 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 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

$$
\begin{array}{rcl}
C_{\text{add}} & = & \emptyset \\
C_{\text{mul}} & = & \{\text{add}\} \\
C_{\text{main}} & = & \{\text{add}, \text{mul}, \text{readin}, \text{writeout}\}
\end{array}
$$

$$
\begin{array}{rcllrcl}
U_{\text{readin}} & = & \{\text{R0} - \text{R12}, \text{SP}, \text{LR}, \text{mem}\} & & D_{\text{readin}} & = & \{\text{R0} - \text{R12}, \text{SP}, \text{LR}, \text{mem}\} \\
U_{\text{writeout}} & = & \{\text{R0} - \text{R12}, \text{SP}, \text{LR}, \text{mem}\} & & D_{\text{writeout}} & = & \{\text{R0} - \text{R12}, \text{SP}, \text{LR}, \text{mem}\} \\
U^{(0)}_{\text{add}} & = & \{\text{R0}, \text{R1}, \text{LR}\} & & D^{(0)}_{\text{add}} & = & \{\text{R0}\} \\
U^{(0)}_{\text{mul}} & = & \{\text{R0} - \text{R5}, \text{SP}, \text{LR}, \text{mem}\} & & D^{(0)}_{\text{mul}} & = & \{\text{R0} - \text{R5}, \text{SP}, \text{LR}, \text{mem}\} \\
U^{(0)}_{\text{main}} & = & \{\text{R0}, \text{R4}, \text{R5}, \text{SP}, \text{LR}, \text{mem}\} & & D^{(0)}_{\text{main}} & = & \{\text{R0}, \text{R1}, \text{R4}, \text{R5}, \text{SP}, \text{LR}, \text{mem}\} \\
U^{(1)}_{\text{add}} & = & U^{(0)}_{\text{add}} & & D^{(1)}_{\text{add}} & = & D^{(0)}_{\text{add}} \\
U^{(1)}_{\text{mul}} & = & U^{(0)}_{\text{mul}} & & D^{(1)}_{\text{mul}} & = & D^{(0)}_{\text{mul}} \\
U^{(1)}_{\text{main}} & = & \{\text{R0} - \text{R5}, \text{SP}, \text{LR}, \text{mem}\} & & D^{(1)}_{\text{main}} & = & \{\text{R0} - \text{R5}, \text{SP}, \text{LR}, \text{mem}\} \\
U^{(2)}_{\text{add}} & = & U^{(1)}_{\text{add}} & & D^{(2)}_{\text{add}} & = & D^{(1)}_{\text{add}} \\
U^{(2)}_{\text{mul}} & = & U^{(1)}_{\text{mul}} & & D^{(2)}_{\text{mul}} & = & D^{(1)}_{\text{mul}} \\
U^{(2)}_{\text{main}} & = & U^{(1)}_{\text{main}} & & D^{(2)}_{\text{main}} & = & D^{(1)}_{\text{main}}
\end{array}
$$

Figure 4. Computing U_f and D_f sets for the functions of the program given in Fig. 1

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 all 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 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. Since the whole memory is represented as a single argument, the definition of the memory in an instruction k is a reaching definition for the use of the memory in another instruction j if there is a path in the CFG from k to j. 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. If a basic block is control dependent on another basic block then it will eventually depend on the last instruction of that block and also on the arguments of that instruction. Hence for all basic block nodes we add control dependence edges coming from the nodes representing the used arguments of the last instructions of basic blocks they depend upon. Similarily, the actual-in parameters of a call site basic block are dependent on the arguments of the last instruction of the block, which is just the function call instruction.

Fig. 5 shows a part of the program dependence graph of the example program. Because of lack of space and for clarity, data dependences are only shown between the arguments of instructions in some selected blocks.

3.2. Improving PDG

Although the PDG built as described in Section 3.1 is safe, it is overly conservative, mainly due to the conservative approach of the data dependence analysis and the lack of use of architecture specific information. In this section we present two approaches for improving the precision of the DDG. One is based on a heuristical analysis of function prologs and epilogs, while the other is a more sophisticated analysis of the memory access of the instructions.

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

If the set of saved and restored registers can be deter-

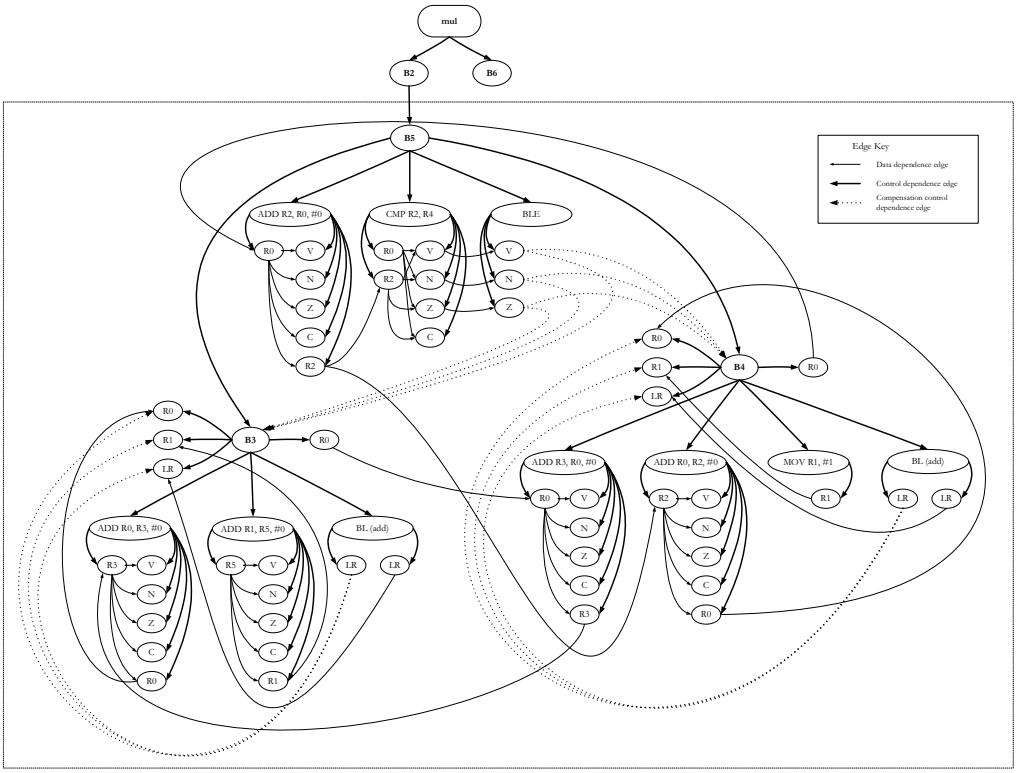

Figure 5. Part of the PDG of function `mul` of the example program presented in Fig. 1. Data dependences are detailed only between blocks B3, B4 and B5.

mined we can redefine the set of output parameters for a function f as $D_f \setminus S_f$, where D_f is defined in Fig. 3, and S_f is the set of registers saved on entry and restored on exit in f. Using the new set of input and output parameters to build the PDG the slice is going to suffer less from imprecision caused by the conservative handling of function calls.

Another source of imprecision is the handling of memory access in data dependence analysis. At the binary level the high-level concepts of variables and function parameters do not exist, so compilers use registers in place of them. But since in most architectures the number of available registers is limited, and registers are also used to store the temporary results of computations in the program. Those parameters and variables that cannot be assigned to registers are usually stored in a specific portion of memory called the stack. Since high-level programs usually make heavy use of variables a more precise analysis of data dependences across the stack can give a notable improvement in slicing accuracy.

In our procedure we characterize 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. The lattice and its elements are shown in Fig. 6.

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

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

The algorithm uses the CFG for propagating information. How is this done? First of all, the first instruction is placed on a worklist. Then a node is chosen and removed from the worklist, and examined. The lattice elements associated with the registers at the entry of the examined instruction become the meet of the lattice elements associated with the corresponding registers at the exit of the preceed-

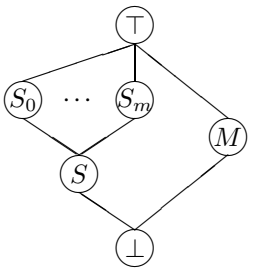

Figure 6. The lattice

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

Figure 7. Rules for $\sqcap$

ing instructions. The meet rules are given in Fig. 7. The instruction is evaluated based on the new input values and then the exit values are determined. If any of the computed exit values differ from the corresponding lattice elements associated with the registers at the exit point of the instruction the following nodes are added to the worklist. The process is repeated until the worklist is empty.

Using the results of this process the data dependence analysis described in Section 3.1 can be improved so as to avoid adding superfluous dependence edges to the graph. In the conservative approach the entire memory was represented by just one argument, but using the results of the above algorithm the used and defined arguments can be determined more precisely. Instead of the argument representing the whole memory we can use arguments labeled according to the same lattice elements as those used in the analysis. This is used to represent certain parts of the memory.

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

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

The results of the improvements are discussed in Section 6.

3.3. Intraprocedural slicing

The PDGs built so far can be used to compute intraprocedural slices by treating call sites as instructions with actual-in and actual-out parameters as used and defined arguments, respectively, where each defined argument is data dependent on all used arguments.

A slice can be computed for any set of used or defined arguments as the slice criterion by traversing via control and data dependence edges [18]. The resulting program slice consists of instruction nodes reached during the graph traversal.

3.4. Interprocedural slicing using 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. The resulting graph is the system dependence graph (SDG) of the program.

Call edges represent dependences between call sites and function entry points. But, as described in Section 3.1, dependences exist in practice between the arguments of the last instructions of the call sites and the function entry points, hence we will add call edges for each function entry node coming from the nodes representing the used arguments of the last instructions of the corresponding call site basic blocks.

Fig. 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.

To finish the SDG we augment it with summary edges to represent dependences between actual-in and actual-out parameters using the algorithm of Reps et al. [19].

The SDG built this way can be used to compute slicies using the two pass algorithm of Horwitz et al. [11] using a set of argument nodes as a slicing criterion.

Fig. 9 shows the interprocedural backward slice of the example program w.r.t. R0 used by the instruction at memory address 00002F2A, using the results of the optimizations presented in Section 3.2. The slice contains the instructions responsible for the loop control logic and the computation of the sum in function `main` and the whole function `add`. *Without the improvements the slice would contain the whole function* `mul` *and all the instructions in basic blocks B8, B10 and B11.*

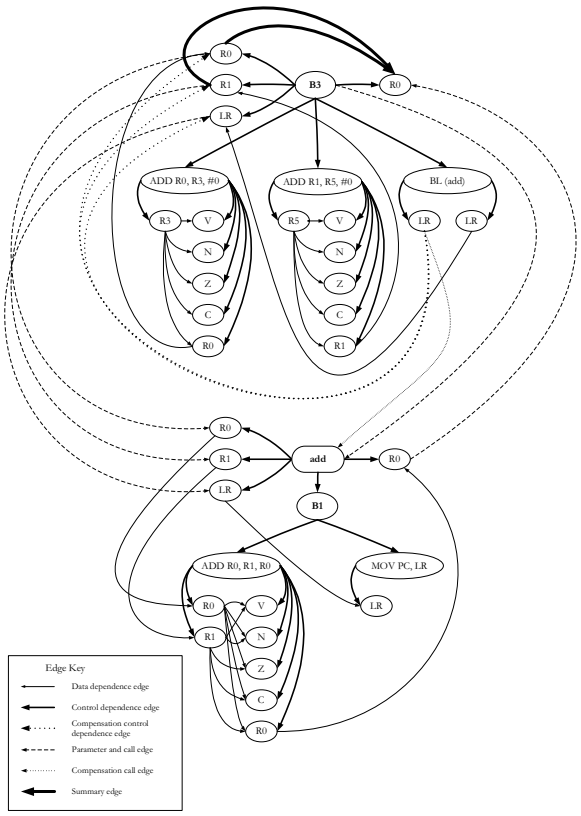

Figure 8. A portion of the SDG of the example program given in Fig. 1, showing the function `add` and the call site B3 in the function `mul`.

```
                        add:
00002ECC    1808        ADD R0, R1, R0          B1
00002ECE    46F7        MOV PC, LR
...
                        main:
00002EFA    B530        PUSH {R4,R5,LR}         B7
00002EFC    B082        ADD SP, #-8
00002EFE    F7FFFFD5    BL 0x00002EAC (readin)
00002F02    1C05        ADD R5, R0, #0          B8
00002F04    2000        MOV R0, #0
00002F06    9000        STR R0, [SP, #0]
00002F08
00002F0A
00002F0C    2401        MOV R4, #1
00002F0E    42AC        CMP R4, R5
00002F10    DA0B        BGE 0x00002F2A
00002F12    9800        LDR R0, [SP, #0]        B9
00002F14    1C21        ADD R1, R4, #0
00002F16    F7FFFFD9    BL 0x00002ECC (add)
00002F1A    9000        STR R0, [SP, #0]        B10
00002F1C
00002F1E
00002F22                                        B11
00002F24    3401        ADD R4, #1
00002F26    42AC        CMP R4, R5
00002F28    DBF3        BLT 0x00002F12
00002F2A    9800        LDR R0, [SP, #0]        B12
00002F2C
00002F2E
00002F32                                        B13
00002F34
```

Figure 9. An interprocedural backward slice of the example program w.r.t. R0 used by the instruction at memory address 00002F2A.

Table 1. Benchmark program size summary

Program	Code	Instructions	Source lines
ansi2knr	11,698	5,835	693
decode	14,594	7,283	1,593
bzip2	27,896	13,934	4,247
toast	32,464	16,218	5,997
sed	38,116	19,044	12,241
cjpeg	88,970	44,468	29,957

4. Experimental results

We implemented our solution and evaluated it on programs taken from the SPEC CINT2000 [21] and MediaBench benchmark suites [15]. 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 programs ranged from 11 to 89 kilobytes (see Table 1). (Our implementation was also able to analyse and slice programs with 400 kilobytes of code but because of time constraints we have not been able to compute slices for enough slicing criteria to display the results in the tables.)

We built the CFG for all the selected programs, performed code and data dependence analyses (both the conservative and improved ones, as described in Sections 3.1 and 3.2) to obtain PDGs for each function and finally created the SDGs. Tables 2 and 3 show the summaries of edge types in the graphs as well as the differences between the conservative and improved approaches. As it can be seen the reduction in the number of data dependence and summary edges in the SDGs are, on average 27% and 32%, respectively, and can be as high as 59% and 36%.

Table 2. CFG edge summary

Program	Control	Call
ansi2knr	1,889	572
decode	2,085	580
bzip2	3,782	1,414
toast	3,344	1,222
sed	6,691	1,678
cjpeg	9,407	3,628

Once we obtained the SDGs for all benchmark programs we selected 5 instructions from every function (only 3 instructions per function from the `cjpeg` program, because of time constraints) and computed intraprocedural slices for their used arguments as slice criteria. As Table 4 shows, the slices contain 31%-58% of instructions on average in the conservative approach and 0%-3% less in the improved approach. Although the improvements in the PDG building process lead to a high reduction in the number of data de-

Table 3. SDG edge summary

Program	Control dependence	Data dependence		Summary	
		conservative	improved	conservative	improved
ansi2knr	1964	66,424	51,470	1,959	1,276
decode	2029	71,216	56,086	1,988	1,274
bzip2	4129	203,308	161,161	5,807	4,075
toast	3459	190,204	135,583	4,118	2,844
sed	7192	795,928	470,836	6,764	4,865
cjpeg	10307	720,381	555,406	10,030	6,408

Table 4. Intraprocedural slicing summary

Program	Criteria	Average size of functions (instructions)	Average size of slices (instructions)	
			conservative	improved
ansi2knr	936	55	20	19
decode	1,066	58	18	16
bzip2	1,549	85	35	34
toast	1,702	75	36	36
sed	2,037	90	43	42
cjpeg	1,361	88	51	50

Table 5. Interprocedural slicing summary

Program	Criteria	Size of program (instructions)	Average size of slices (instructions)	
			conservative	improved
ansi2knr	936	5,835	3,604	3,413
decode	1,066	7,283	4,121	3,874
bzip2	1,549	13,934	7,741	7,494
toast	1,702	16,218	9,019	8,718
sed	2,037	19,044	11,580	11,410
cjpeg	1,361	44,468	30,186	29,896

pendence edges, the reduction in slice size is only moderate. This moderate improvement is due to the conservative handling of memory access in called functions, which cancels out the effects of the improvements.

We also computed interprocedural slices using the constructed graphs w.r.t. the same slicing critera as in the intraprocedural case. Table 5 shows the results of the computations. We obtained slices that had, on average 56%-68% of all the instructions using the conservative approach and 1%-4% fewer instructions using the improved one. According to our investigations the moderate improvement in the size of interprocedural slices is mainly attributable to two factors: first, the conservative handling of memory access of the called functions and, second, the high number of unresolved function calls.

5. Related work

The slicing of binary executables requires building a CFG from raw binary data. Debray et al. make use of a technique similar to the one outlined above in their code compaction solution [9] to build a CFG for binaries compiled for the Alpha architecture.

To our knowledge there are no practical interprocedural slicing solutions for binary executable programs and useful intraprocedural binary slicing is hard to find in the literature as well. Larus and Schnarr use intraprocedural static slicing in their binary executable editing library called EEL [14]. They use slicing to improve the precision of control flow analysis in the case of indirect jumps mostly occurring in compiled form of case statements. With the help of backward slicing they are able to analyse such constructs in an architecture and compiler-independent manner.

Cifuentes and Fraboulet also use intraprocedural slicing

for solving indirect jumps and function calls in their binary translation framework [8]. Bergeron et al. in [4] suggest to use interprocedural static slicing for analysing binary code to detect malicious behavior. The computed slices should be verified against behavioral specifications to statically detect potentially malicious code. They did not discuss the potential problems of analysis binary executables neither presented any experimental result.

6. Conclusion and future work

In this paper we described how interprocedural slicing can be applied to binary executables. We presented a conservative dependence graph based approach and also improvements. We evaluated both approaches on programs compiled for ARM architecture and achieved an interprocedural slice size of 56%-68% in average using the conservative approach and 1%-4% reduction using the improvements. The moderate improvements are due to the conservative handling of memory and indirect function calls.

Currently we are working to make the interprocedural slicing of binary executables more accurate. We focus on extending the current solution to allow propagation of information on stack and memory access across function boundaries and make the SDG and the slices more precise.

In the future we also plan to handle the problem presented by unresolved function calls by making the call graph of the analysed programs more accurate using profiling information. Another interesting task for the future would be to apply the solution presented above on Java bytecode programs.

References

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

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

[3] J.-F. Bergeretti and B. Carré. Information-flow and data-flow analysis of **while**-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, Jan. 1985.

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

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

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

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

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

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

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

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

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

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

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

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

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

[17] Microsoft Corporation. Microsoft Portable Executable and Common Object File Format Specification Version 6.0, Feb. 1999.

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

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

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

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

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

[23] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Version 1.2, May 1995.

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