Interprocedural Static Slicing of Binary Executables

Ákos Kiss, Judit Jász, Gábor Lehotai and Tibor Gyimóthy
Research Group on Artificial Intelligence
University of Szeged and Hungarian Academy of Sciences
Aradi vénétanúk 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 Frabuolo 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-
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 have 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. Similarly, there is no standard way of analysing the behaviour of the instructions running on different platforms. Although 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 constructs 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 nodes, 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.

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

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 instructions 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 $f$.

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

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

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

Figure 3. Computing $U_f$ and $D_f$
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. Similarly, 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. Similarly, 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.

...
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 slices 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.
4. Experimental results

We implemented our solution and evaluated it on programs taken from the SPEC CINT2000 [21] and Media-Bench 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%.

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 depen-
Table 3. SDG edge summary

<table>
<thead>
<tr>
<th>Program</th>
<th>Control dependence</th>
<th>Data dependence</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>conservative</td>
<td>improved</td>
<td>conservative</td>
</tr>
<tr>
<td>ansi2knr</td>
<td>1964</td>
<td>66,424</td>
<td>51,470</td>
</tr>
<tr>
<td>decode</td>
<td>2029</td>
<td>71,216</td>
<td>56,086</td>
</tr>
<tr>
<td>bzip2</td>
<td>4129</td>
<td>203,308</td>
<td>161,161</td>
</tr>
<tr>
<td>toast</td>
<td>3459</td>
<td>190,204</td>
<td>135,583</td>
</tr>
<tr>
<td>sed</td>
<td>7192</td>
<td>795,928</td>
<td>470,836</td>
</tr>
<tr>
<td>cjepg</td>
<td>10307</td>
<td>720,381</td>
<td>555,406</td>
</tr>
</tbody>
</table>

Table 4. Intraprocedural slicing summary

<table>
<thead>
<tr>
<th>Program</th>
<th>Criteria</th>
<th>Average size of functions (instructions)</th>
<th>Average size of slices (instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>conservative</td>
<td>improved</td>
</tr>
<tr>
<td>ansi2knr</td>
<td>936</td>
<td>55</td>
<td>20</td>
</tr>
<tr>
<td>decode</td>
<td>1,066</td>
<td>58</td>
<td>18</td>
</tr>
<tr>
<td>bzip2</td>
<td>1,549</td>
<td>85</td>
<td>35</td>
</tr>
<tr>
<td>toast</td>
<td>1,702</td>
<td>75</td>
<td>36</td>
</tr>
<tr>
<td>sed</td>
<td>2,037</td>
<td>90</td>
<td>43</td>
</tr>
<tr>
<td>cjepg</td>
<td>1,361</td>
<td>88</td>
<td>51</td>
</tr>
</tbody>
</table>

Table 5. Interprocedural slicing summary

<table>
<thead>
<tr>
<th>Program</th>
<th>Criteria</th>
<th>Size of program (instructions)</th>
<th>Average size of slices (instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>conservative</td>
<td>improved</td>
</tr>
<tr>
<td>ansi2knr</td>
<td>936</td>
<td>5,835</td>
<td>3,604</td>
</tr>
<tr>
<td>decode</td>
<td>1,066</td>
<td>7,283</td>
<td>4,121</td>
</tr>
<tr>
<td>bzip2</td>
<td>1,549</td>
<td>13,934</td>
<td>7,741</td>
</tr>
<tr>
<td>toast</td>
<td>1,702</td>
<td>16,218</td>
<td>9,019</td>
</tr>
<tr>
<td>sed</td>
<td>2,037</td>
<td>19,044</td>
<td>11,580</td>
</tr>
<tr>
<td>cjepg</td>
<td>1,361</td>
<td>44,468</td>
<td>30,186</td>
</tr>
</tbody>
</table>

We also computed interprocedural slices using the constructed graphs w.r.t. the same slicing criteria 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 interprocedural 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