Intraprocedural Static Slicing of Binary Executables*

Cristina Cifuentes Antoine Fraboulet
Centre for Software Maintenance
Department of Computer Science
University of Queensland
Brisbane Qld 4072, Australia
{cristina,afrab}@cs.uq.edu.au

Abstract

Program slicing is a technique for determining the set of statements of a program that potentially affect the value of a variable at some point in the program. Intra and interprocedural slicing of high-level languages has greatly been studied in the literature; both static and dynamic techniques have been used to aid in the debugging, maintenance, parallelization, program integration, and dataflow testing of programs.

In this paper we explain how to apply conventional intraprocedural static analysis to binary executables, for the purposes of static analysis of machine-code and assembly code, such as debugging code and determining the instructions that affect an indexed jump or an indirect call on a register. This analysis is useful in the decoding of machine instructions phase of reverse engineering tools of binary executables, such as binary translators, disassemblers, binary profilers and binary debuggers.

1 Introduction

It is well known that a series of programmers worked for days on the manual disassembly of the Internet Worm program in order to determine what the program was doing and how to prevent it from spreading further around the world [10]. It is also well known amongst programmers how hard and time-consuming it is to debug a binary executable program that needs to be patched (i.e. bug fixed) when the source-code to that program has been lost or is no longer supported by its authors. Both these cases would take less time if automated debugging tools for executable programs were available – at present only simple disassemblers are available to programmers. Further, recent reverse engineering tools of binary executables, such as static binary translators [23], rapidly translate/migrate binary executables from one platform to another without using slicing techniques, hence relying on a runtime interpreter to translate any code that was not translated statically. Slicing techniques can be used to reliably determine the range of values of indexed jump tables and calls that depend on registers, rather than using pattern matching of the code against a set of code skeletons (generated from different compilers).

Program slicing is a technique for automatically decomposing high-level programs by analysing its control and data flow information. This technique was developed by Weiser [28] after he realized that programmers tend to slice programs in their head when debugging in order to understand complex code [27]. Static slicing techniques aim at extracting a minimal program that captures the behaviour of a source-code program with respect to a specified variable and location; this problem is solved by a reachability analysis. Over the years, static slicing has been applied to debugging, maintenance, and parallelization of source code, as well as program comprehension.

Throughout this paper, the terms *binary executable* and *executable program* are used as synonyms to refer to the program generated by the compilation/linkage process; i.e. a machine-dependent program written in machine code. Executable files do not normally contain symbol table information, hence our analysis does not assume use of such information.

1.1 The Need for Tools for Executable Programs

Reverse engineering tools of binary executables include disassemblers, binary translators, decompilers, binary profilers, and binary debuggers. These tools share two phases in common: they need to decode the information stored in the binary file, and they need to decode the machine instructions and translate them to an assembly representation or an equivalent interme-

*This work is partly supported by the Australian Research Council under grant No. A49702762 and Sun Microsystems Labs.

diate representation. The main problem in the decoding of machine instructions is the separation of data and code; they are indistinguishable, and hence this problem is equivalent to the halting problem [14, 18]. Nevertheless, approximations to the solution are possible in practice.

Given the entry point to a program, machine instructions are decoded following all possible paths in the code. Statically, this algorithm cannot determine what paths to traverse next when faced with an indexed jump instruction, or an indirect call or jump instruction on the value of a register. That is, statically we do not know the value of the register, hence we cannot determine the target address of the jump or call. For example, the indexed jump in Figure 1 at instruction 71 depends on the contents of register `bx`. Its contents is unknown statically but its range of values can be known by performing a backward slice on register `bx` at instruction 71, and analyzing that slice for upper and lower bounds that the register can take. Clearly, slicing techniques can aid in the recovery of this problem, for both, indexed and indirect register-based instructions, hence allowing for a way of possibly determining all paths in a given program. Once all relevant machine instructions are decoded, the code can be translated to assembly or any other intermediate representation, and the reverse engineering tool at hand can do transformations on the intermediate representation to produce its result (i.e. an assembly program in the case of a disassembler, a binary program for a different target machine in the case of a binary translator, a high-level language program in the case of a decompiler, and so on).

```
[...]
038 L6:  XOR            ah, ah
039      MOV            dx, ax
040      MOV            bx, ax
041      SUB            bl, 20h
042      CMP            bl, 60h
043      JAE            L7
044      MOV            bl, [bx+291h]
045      CMP            bx, 17h
046      JBE            L8
047      JMP            L9
048 L9:  MOV            si, [bp-10h]
049      MOV            di, [bp-4]
050      MOV            al, 25h
[...]
070 L8:  SHL            bx, 1
071      JMP   word ptr cs:[bx+9B3h]
[...]
```

Figure 1: Extract Assembly Code with Indexed Jump Statement

As previously mentioned, executable software can have bugs as the methods used for software testing (e.g. program inspections, mathematically-based verification, defect testing, and cleanroom software development) do not necessarily remove all possible bugs from a given program [24]. This means that once you buy a program and run it, if there is some defect with it, you need to report the bug to the software developers and possibly pay for a new version of the bug-free software. However, if the developers are not available to support this software (i.e. they do not want to support the software anymore, or they are out of business), the user is the only one that can make the change(s). A debugging tool for executable programs is what the user would need — if only one had been developed. Also, the analysis of worm or virus code is an area that would benefit considerably from a debugger of executable programs, as once an area of code is flagged to be critical, a slice of that code could be determined to simplify the process of debugging the thousands of machine instructions in the binary executable. So far, no tool for the debugging of executable programs is available, only "dumb" disassemblers that disassemble machine instructions given a particular memory address, but do not trace the machine code and construct the control flow graph of the program, or slice the program based on a given criterion.

It is also known amongst the maintenance community that the only way to make changes to some pieces of legacy binary code is by way of patching the binary file itself. The development of debugging tools for real, non-trivial executable programs has not been addressed in the literature and is not a trivial problem.

In this paper we look at the application of conventional static intraprocedural slicing techniques to binary executables. We list the required changes in order to apply these techniques to assembly code. We are interested in integrating precise slicing techniques in tools that manipulate machine-code, when analyzing indexed jumps and calls that depend on the contents of a register.

2 Previous Work

A substantial amount of work has been reported in the literature on static slicing techniques. Three complete surveys of this area have recently been reported in the literature [17, 26, 4], providing for a complete summary of slicing techniques and a comparative classification of such techniques. We report here on the relevant static techniques for assembly-like languages.

Slicing techniques were initially developed by Weiser [27, 28], who made use of a slicing criterion

and a set of equations to determine which statements to include in a slice. Horwitz et al [15, 16] used and extended the concept of the program dependence graph (PDG), developed by Ferrante et al [11], to create precise intraprocedural slices for single-procedure *structured* programs. They developed the concept of the system dependence graph (SDG) to produce precise interprocedural slices, by taking into account the calling context of each procedure.

Three techniques were developed to deal with *unstructured* programs that make use of the `goto` statement. High-level languages such as C and Fortran make use of this statement. Ball and Horwitz [3] and Choi and Ferrante [5] developed similar techniques to deal with arbitrary flow of control, by requiring the control dependence graph (CDG) to be constructed from an augmented control flow graph (CFG) but the corresponding data dependence graph (DDG) to be constructed from the original CFG. Agrawal [1] created a better method to deal with `goto` statements which does not modify the CFG or the SDG. He relied on the postdominator tree (PDT) (created during the construction of the SDG) and the lexical successor tree (LST) to determine which jumps to add to the final slice, hence adding a few steps to the conventional slicing algorithm by Horwitz et al [16] without modifying existing graphs used in debugging.

In the area of binary executables, static slicing techniques have been applied to the binary profiler qpt [19] which rewrites RISC executable files to measure program behavior. The techniques have been embedded in EEL [20], an executable editing library for RISC-based binaries, as part of the decoding phase of machine instructions.

3 Representation of Binaries

A binary executable is the machine-code version of a high-level or assembly program which has been compiled (or assembled) and linked for a particular platform (M, OS) (i.e. for a particular machine M and operating system OS). The general format of a binary executable varies widely based on the binary-file format used by the OS; the EXE [9] format (used by the DOS operating system) is very simple and only includes a program header and the image of both the code and data of the program, whereas the ELF [25] format (used by the multiplatform operating system Solaris) includes comprehensive information as to the different sections of the program; including a program header, section header, relocation table, symbol table, string table, dynamic symbol information, data segment, text segment, and more. When running a program, the binary-file format is decoded by the operating system's loader, which loads the program into memory and passes control to the program via its entry point.

Our test vehicle is `dasm`, the disassembler of the `dcc` decompiler [8], which implements a decoder of the EXE format, a decoder of 80286 machine instructions, and creates an intermediate representation of the program in terms of the program's call graph, CFG of each procedure based on basic blocks[1], and low-level icode instructions [6]. We have removed from `dasm` all pattern matching and heuristics used to deal with indexed and indirect instructions. Hence, we perform a conservative traversal of the machine instructions without traversing unknown targets of transfers of control such as indirect calls on a register or indexed jumps on a register.

The low-level icode instructions resemble assembly instructions of the machine, but have the property of only performing one operation at a time. For example, the 80286 `DIV bx` assembly instruction returns the result of the division of `dx:ax` by `bx` in register `ax`, and its modulus in register `dx`. Clearly, if the division and modulus operations were performed sequentially without the use of a temporary storage location, register `ax` would be overwritten prior to the modulus operation. We represent such instruction as 3 low-level icode instructions making use of an extra register `tmp`:

```
mov tmp, dx:ax    ; tmp = dx:ax
div tmp, bx       ; ax = tmp / bx
mod tmp, bx       ; dx = tmp % bx
```

In this way, the 250 machine instructions of the 80286 were mapped to 110 low-level icode instructions. We continue to use the 80286 as a test vehicle due to its CISC characteristics and its manageable size; the Pentium has over 500 machine instructions. For the purposes of this analysis, the Pentium does not add extra complexity to the problem, only extra maintenance time.

As part of decompilation analysis to recover high-level code from this intermediate representation, data flow analysis is performed in the CFGs to recover high-level expressions, remove dead code, recover parameters to procedures, remove any references to registers and condition codes, and remove references to the stack (either local variables or parameters) [6]. Part of this data flow analysis will be required for binary translation purposes, and is required for slicing purposes; the relevant parts will be mentioned in Section 4 during the explanation of the slicing algorithm.

[1] A basic block is a sequence of instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

4 Slicing of Binary Executables

For slicing purposes, we assume we have a disassembler such as `dasm` which not only decodes the binary file and its machine instructions, but also stores the instructions in terms of low-level instructions that resemble assembly code and control flow graphs for each procedure. Such intermediate representation allows us to analyze the graphs as per standard compiler optimization techniques.

Throughout this paper we refer to Horwitz et al [16]'s algorithm as the *conventional static slicing* algorithm, and the extensions performed by Agrawal [1] as the *conventional goto* algorithm. This latter algorithm is a suitable basis for slicing machine-code and assembly programs but requires some changes as described in 4.1 and 4.2. Figure 2 shows the disassembly of the `main` procedure of the test C program in Figure 3.

The conventional goto algorithm uses CFGs of instructions, the PDG to hold data and control dependencies, the PDT to construct the PDG, and the lexical successor tree (LST) to add the missing (unconditional) jumps to the slice. A statement S' is said to be the immediate lexical successor of statement S if deleting S from the program causes the control to pass to S' whenever it reaches the corresponding location in the new program [1].

When slicing binary code, we use CFGs of basic blocks, CDG, PDT, and use-definition (ud) chains as per the following description. We do not construct the LST as this structure is not adequate for all assembly programs, as explained in 4.2.

The steps of the algorithm are as follows:
1 Determine the slice using the conventional algorithm,
2 Add unconditional jumps and returns to the slice,
3 Fix jump labels.

4.1 Step 1

Traditional slicing techniques use individual statements as the node granularity in graphs. When slicing machine-code and assembly representations, using basic blocks as the node granularity in the graph is more appropriate as the number of instructions tends to be fairly large when compared to their high-level language counterpart. A study on the reduction rate of assembly instructions to high-level statements on a CISC machine, when being regenerated from the assembly code, gave an average of over 70% of the assembly instructions being eliminated by the analysis [6].

Control dependencies have traditionally being built based on the PDT of a procedure, as it shows nodes in the path to the end of the routine satisfying postdominance relationship. These control dependencies are

```
                main  PROC  NEAR
000     PUSH            bp          ; ud(bp)=-1
001     MOV             bp, sp      ; ud(sp)=-1
002     SUB             sp, 2       ; ud(sp)=-1
003     PUSH            si          ; ud(si)=-1
004     PUSH            di          ; ud(di)=-1
005     XOR             si, si
006     MOV  word ptr [bp-2], 0
007     MOV             di, 0FFFBh
008     JMP             L1
009 L1: CMP             di, 0Ah     ; ud(di)=7
010     JL              L2          ; ud(cc)=9
011     PUSH            si          ; ud(si)=5,32,50,56
012     MOV             ax, 194h
013     PUSH            ax          ; ud(ax)=12
014     CALL near ptr printf
015     POP             cx
016     POP             cx
017     PUSH word ptr [bp-2]
018     MOV             ax, 19Eh
019     PUSH            ax          ; ud(ax)=18
020     CALL near ptr printf
021     POP             cx
022     POP             cx
023     POP             di
024     POP             si
025     MOV             sp, bp      ; ud(bp)=1
026     POP             bp
027     RET
028 L2: OR              di, di      ; ud(di)=7,34
029     JG              L3          ; ud(cc)=28
030     MOV             ax, si      ; ud(si)=32,50,56,5
031     ADD             ax, di      ; ud(ax)=30  ud(di)=7,34
032     MOV             si, ax      ; ud(ax)=31
033     JMP             L4
034 L4: INC             di          ; ud(di)=7,34
035     JMP             L1
036 L3: MOV             ax, [bp-2]  ; ud([bp-2])=6,38
037     INC             ax          ; ud(ax)=36
038     MOV             [bp-2], ax  ; ud(ax)=37
039     MOV             ax, di      ; ud(di)=7,34
040     MOV             bx, 2
041     CWD                         ; ud(ax)=39
042     MOV             tmp, dx:ax  ; ud(dx:ax)=41
043     IDIV            bx          ; ud(bx)=40  ud(tmp)=42
044     MOD             bx          ; ud(bx)=40  ud(tmp)=42
045     OR              dx, dx      ; ud(dx)=43
046     JNE             L5          ; ud(cc)=45
047     MOV             ax, di      ; ud(di)=7,34
048     SHL             ax, 1       ; ud(ax)=47
049     ADD             ax, si      ; ud(si)=5,32,50,56 ud(ax)=48
050     MOV             si, ax      ; ud(ax)=49
051     JMP             L4
052 L5: MOV             ax, di      ; ud(di)=7,34
053     MOV             dx, 3
054     MUL             dx          ; ud(dx)=53  ud(ax)=52
055     ADD             ax, si      ; ud(si)=32,50,56  ud(ax)=54
056     MOV             si, ax      ; ud(ax)=55
057     JMP             L4
                main  ENDP
```

Figure 2: Disassembled Code for main() Procedure of Test Program, Annotated with Use-Definition Chains.

hard to determine without constructing the PDT in machine-code and assembly programs as a procedure may have more than one return instruction and therefore there are different paths to the end of the routine. Also, we cannot assume that the nesting level of the statements in the procedure can be used as the procedure's graph may be unstructured in general and such

```
void main()
{ int sum, positives, x;
    sum = 0;
    positives = 0;
    for (x = -5; x < 10; x++) {
        if (x <= 0)
            sum = sum + x;
        else {
            positives = positives + 1;
            if (x % 2 == 0)
                sum = sum + 2 * x;
            else
                sum = sum + 3 * x;
        }
    }
    printf ("sum = %d\n", sum);
    printf ("positives = %d\n", positives);
}
```

Figure 3: Sample Test C Program.

assumption would restrict the techniques to a limited number of binary executables. Note however that during control flow analysis of the CFG for the purposes of recovering the underlying control structures of a graph and their nesting level, unstructured graphs can be forced to have nesting levels based on a particular criteria [7] or be node-split to replicate nodes and convert the graph into a structured one [13]. In the present case though, it is not worth the extra analysis to determine this imposed nesting level, so the contruction of the control dependence graph (CDG) based on the PDT and the CFG is favoured. The PDT can be efficiently constructed in $O(N\alpha(N))$ time, where N is the number of nodes in the CFG [21]; a simpler implementation is described in [12]. The CDG can be built in N^2 time by walking the PDT [11]. The CFG, PDT and CDG for the test program are shown in Figure 4. The nodes entry, start and end are traditionally used in the analysis by augmenting the CFG with them; these nodes need not necessarily exist in the CFG.

We represent data dependencies of the program in terms of ud-chains [2] at the procedure level rather than DDGs as these chains are useful to other types of analyses that can be performed in the binary code (e.g. dead register elimination, dead condition code elimination). The ud-chains are generated for each register and condition code used in an instruction. However, we first perform idiom analysis of the icode instructions so that an instruction such as `xor si,si` (instruction 5 in Figure 2) is treated as `mov si,0` which is the end effect of xor'ing a register to itself. In this code, the compiler decided to generate an `xor` instruction rather than a `mov` instruction as the former takes less machine cycles. Failure to perform this analysis would assume a (redundant) use of register `si` prior to its re-definition, hence creating an unnecessary dependency on its interprocedural (caller's) `si` value.

In order to cater for conditional jumps that are not dependent on a register (e.g. jump on less than (`jl`), jump on not equal to zero (`jne`), as opposed to jump if `cx` register zero (`jcxz`)), use-definition chains on condition codes need to be used. However, given that a large percentage of instructions in CISC machines define and use condition codes, dead-condition code elimination has to be performed first; this analysis is consistent with decompilation and binary translation analyses required to determine the effects of machine instructions, as reported in [6]. For example, in the following code

```
cmp dx, bx   ; dF = {CF,ZF,SF}
jg L10       ; uF = {SF}
```

the compare instruction `cmp dx,bx` sets the condition codes CF (carry flag), ZF (zero flag) and SF (sign flag). The conditional jump instruction `jg L10` uses the sign flag to determine whether the jump is to be taken or not. Dead-condition code elimination would determine that the zero and carry flag set by the compare instruction are dead and therefore irrelevant to the analysis; however, the sign flag is set by this instruction and then used in the conditional jump, hence making the conditional jump data dependent on the comparison instruction. Figure 2 shows use-definition chains at the procedure level for all registers and condition codes in the `main` procedure. In these chains, a -1 denotes that the register was set outside the current procedure, and `cc` denotes the relevant condition code(s) used by the conditional jump after performing dead-condition code elimination.

Overall, condition codes need to be analyzed for all instructions and extraneous ones removed via dead-condition code elimination. In the example of Figure 2, only conditional jumps use these condition codes, however, Figure 6 shows the dependency of instruction `LODSB` on the direction flag, previously cleared by instruction `CLD` (i.e. inter-basic block analysis is needed in some cases).

4.2 Steps 2 and 3

The second step adds unconditional jump and return instructions. In the conventional goto algorithm, the lexical successor tree is used to represent the next high-level statement at the same nesting level of a given statement; the lexical successor of the last instruction is the end of the procedure (i.e. } in C). The

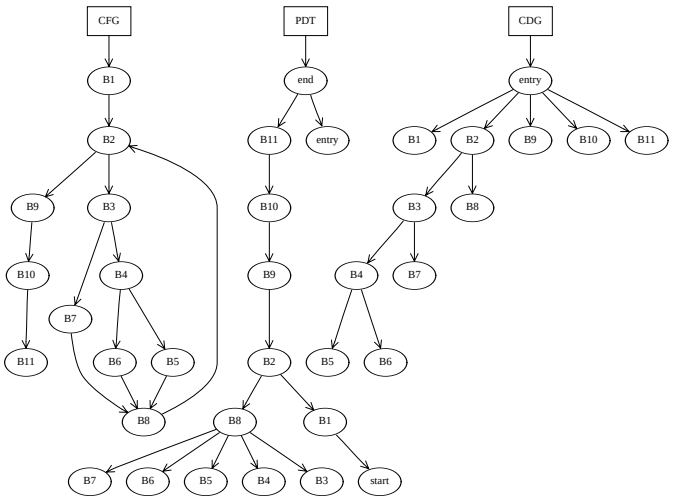

Figure 4: Control Flow Graph, Program Dependence Tree, and Control Dependence Graph for the Program of Figure 2.

lexical successor is well-defined in high-level language programs; a slicing algorithm need work only with *one* representation of the input program. However, in binary code, there is more than one representation in assembly code of the same binary code. This variety of representations leads to different lexical successors based on the representation of the same binary code. For example, in Figure 2, if the unconditional jump at instruction 33 is removed, the control passes to instruction 34, hence the lexical successor of 33 is 34. However, if the unconditional jump at instruction 57 is removed, we cannot determine where the control passes to, given that the end of the procedure is not *at the end* but interleaved within the code at instruction 27.

An unconditional jump and a return instruction introduce a break in the flow of control of the program. Adding these instructions to the slice requires that such instructions belong to basic blocks that were traversed in the construction of the modified conventional algorithm (step 1). This step is linear on the number of basic blocks.

The last step fixes target labels by checking all the jumps that belong to the slice.

Applying this modified version of the conventional goto algorithm creates precise static slices for machine-code and assembly code. Figure 5 shows the slice obtained for the test program of Figure 2 when slicing on register `si` at instruction 11.

5 Experience

We implemented the above mentioned techniques on `slicer286`, as an extension of the 80286 `dasm` disassembler for DOS binaries. `slicer286` disassembles binaries and displays each procedure at a time, prompting the user whether he wants to perform slicing on the current procedure. Slicing is allowed on any register that the machine has, at any instruction in the procedure (instructions are enumerated on the left-hand side for use convenience). Figures 5 and 6 were generated by this tool.

```
                main  PROC  NEAR
005             XOR           si, si
007             MOV           di, 0FFFBh
009     L1:     CMP           di, 0Ah
010             JL            L2
011     Start:
028     L2:     OR            di, di
029             JG            L4
030             MOV           ax, si
031             ADD           ax, di
032             MOV           si, ax
034     L3:     INC           di
035             JMP           L1
036     L4:
039             MOV           ax, di
040             MOV           bx, 2
041             CWD
042             MOV           tmp, dx:ax
044             MOD           bx
045             OR            dx, dx
046             JNE           L5
047             MOV           ax, di
048             SHL           ax, 1
049             ADD           ax, si
050             MOV           si, ax
051             JMP           L3
052     L5:     MOV           ax, di
053             MOV           dx, 3
054             MUL           dx
055             ADD           ax, si
056             MOV           si, ax
057             JMP           L3
                main  ENDP
```

Figure 5: Slice of Test Program with Respect to Register si at Instruction 11.

As can be seen from the sample slices, when slicing on registers at the binary level, a reasonable amount of instructions are returned in the slice. Given that registers are always going to be set by stack variables (locals or arguments), memory locations (globals), or constants, their slices are bound to be a fraction of the code so long as non-aliased memory locations are used. In our analysis we did not include any analysis of aliases (i.e. pointers) as this analysis requires interprocedural analysis which is beyond the scope of the application of these techniques. As mentioned in the introduction, we are interested in solving indexed jumps and indirect calls on registers in a reliable way,

hence if unsure about memory locations, stop at such locations rather than determine their dependencies.

The techniques can clearly be extended to deal with stack variables at no extra cost; notably only use-definition chains on these variables need to be introduced in the analysis (all other data structures already hold the right information). This would be necessary if implementing a debugging slicer tool for example.

Figure 6 showed a dependency of `LODSB` at instructions 15 and 27 on instruction 10 (`CLD`). It was interesting to see that ud-chains on condition codes provided dependencies across basic block boundaries. This type of dependency is not normally expected from user written code, as the compiler normally sets a condition code and uses it within the same basic block (e.g. conditional jumps on extended basic blocks). However, library code may have been written in assembler and hence this type of dependencies may be found.

When slicing high-level language programs, once arrays and pointers are introduced, the returned slices are quite large (almost the whole routine). In comparison, at the binary level, one is normally only interested in base data types such as bytes and words, as this is the only data type information that we can assume comes with the binary. Higher-level information such as arrays are normally not part of the binary, and recovering such information requires extensive analysis and is more inline with decompilation analysis. However, the use of slicing techniques at the compound data type level seems unnecessary.

5.1 Application to the Indexed Jump Example

Applying the described slicing process, the indexed jump on register `bx` at instruction 71 in Figure 1 generates the slice in Figure 6. Instruction `CLD` clears the direction flag, which is later used by instruction `LODSB` to assign to register `al` the contents of the address pointed to by `si`, and increment this address. If the direction flag was set, register `si` would be decremented each time `LODSB` was invoked.

Range-value propagation analysis [22] of this slice would determine that the possible range of values for register `bx` at instruction 71 are [0..17h] words (2 bytes for this particular machine), hence allowing the decoding of the target branches of this switch/case statement. Note however that the slice is lengthier than required for the range-value analysis in this case as instruction 44 depends on the contents of `bx` and as such the slice has included all statements that impinge on that value. Even without knowledge of such memory value, the final comparison restricts register `ax` to be within 0 and 17h words. Hence, we can stop at memory locations during our slicing analysis if this is to be integrated with the detection of jump targets for indexed jumps.

```
001       MOV             bp, sp
010       CLD
014       MOV             si, [bp+6]
015 L1:   LODSB
016       OR              al, al
017       JE              L4
018       CMP             al, 25h
019       JE              L3
020 L2:
026 L3:
027       LODSB
028       CMP             al, 25h
029       JE              L2
038       XOR             ah, ah
040       MOV             bx, ax
041       SUB             bl, 20h
042       CMP             bl, 60h
043       JAE             L3
044       MOV             bl, [bx+291h]
045       CMP             bx, 17h
046       JBE             L5
048 L3:
055 L4:
070 L5:   SHL             bx, 1
071 S:    JMP   word ptr cs:[bx+9B3h]
```

Figure 6: Slice for the Program of Figure 1 on Register bx at Instruction 71.

6 Conclusions

In this paper we have explained how to apply conventional slicing techniques and required changes to slice machine-code and assembly-type languages. The techniques are suitable for the debugging of machine-code and to determine the statements that a particular indexed register is dependent on for the purposes of knowing the range of values that such register can take when analyzing code statically.

For each program, we make use of the control flow graph (CFG) of basic blocks for each procedure as part of the intermediate representation. We determine data dependencies using use-definition chains on registers and condition codes, at the procedure level (after performing an idiom analysis to better represent the semantics of some machine instructions). Condition code dependencies allow us to include for example, unconditional jumps and comparison instructions in the slice. Control dependencies are determined by constructing the control dependence graph for unstructured graphs and post dominator tree (PDT) for each procedure. Based on the PDT and the CFG, unconditional jumps and returns are added to the final slice

if they are in the path to the instructions in the slice. The final slice is non-executable as it does not include all return statements needed in assembly code.

7 Future Work

We are working on a retargetable binary translation framework. The intraprocedural slicing technique described in this paper will be extended to deal with the determination of the range of values that a particular register can have at an indexed jump or call. For more information on the binary translation project check:
`http://www.cs.uq.edu.au/groups/csm/ bintrans.html`

Acknowledgements

We would like to thank David Ung for an initial implementation of the slicing techniques, and the anonymous referees for their comments in ways of improving this paper. The graphs in this paper were created using graphviz.

References

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

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[3] T. Ball and S. Horwitz. Slicing program with arbitrary control-flow. In P.A. Fritzson, editor, *Proceedings of the International Workshop on Automated and Algorithmic Debugging*, Lecture Notes in Computer Science 749, pages 206–222, Linköping, Sweden, May 1993. Springer-Verlag.

[4] D. Binkley and K. Gallagher. *Advances in Computers*, volume 43, chapter Program Slicing. Academic Press, San Diego, CA, 1996.

[5] J. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[6] C. Cifuentes. Interprocedural dataflow decompilation. *Journal of Programming Languages*, 4(2):77–99, June 1996.

[7] C. Cifuentes. Structuring decompiled graphs. In T. Gyimóthy, editor, *Proceedings of the International Conference on Compiler Construction*, Lecture Notes in Computer Science 1060, pages 91–105, Linköping, Sweden, 24-26 April 1996. Springer Verlag.

[8] C. Cifuentes and K.J. Gough. Decompilation of binary programs. *Software – Practice and Experience*, 25(7):811–829, July 1995.

[9] R. Duncan. *The MSDOS Encyclopedia*, chapter 4, pages 107–147. Microsoft Press, 1988.

[10] T. Eisenberg. The computer worm: A report to the provost of Cornell University on an investigation by the commission of preliminary enquiry. Report, Cornell University, Ithaca, USA, 6 February 1989.

[11] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[12] M. Hecht and J. Ullman. A simple algorithm for global data flow analysis problems. *SIAM Journal of Computing*, 4(4):519–532, December 1975.

[13] M.S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, Inc, 52 Vanderbilt Avenue, New York, New York 10017, 1977.

[14] R.N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1979.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 35–46, Atlanta, June 1988. ACM Press.

[16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[17] M. Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems Software*, 31:197–214, 1995.

[18] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[19] J.R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software – Practice and Experience*, 24(2):197–218, February 1994.

[20] J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *SIGPLAN Conference on Programming Languages, Design and Implementation*, pages 291–300, June 1995.

[21] T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[22] J.R.C. Patterson. Accurate static branch prediction by value range propagation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 67–78, La Jolla, California, June 1995. ACM Press.

[23] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[24] I. Sommerville. *Software Engineering*. Addison-Wesley, Reading, Massachusetts, fifth edition, 1996.

[25] SunSoft. *Linker and Libraries Guide – Solaris 2.4*. Sun Microsystems Inc., Mountain View, California, August 1994.

[26] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[27] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[28] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.