

Chapter 2

Taxonomy of Program Features

Abstract All programs have common features and abstractions which are used to create birthmarks. Features can be divided into syntactic and semantic groups. Syntactic features concern themselves with program structure and program form. Semantic features examine the meaning of the program. In this chapter we examine those syntactic and semantic features of programs. Syntactic Features include: (1) Raw Code, (2) Abstract Syntax Trees, (3) Variables, (4) Pointers, (5) Instructions, (6) Basic Blocks, (7) Procedures, (8) Control Flow Graphs, (9) Call Graphs, and (10) Object Inheritances and Dependencies. Semantic features include: (1) API Calls, (2) Data Flow, (3) Procedure Dependence Graphs, and (4) System Dependence Graphs.

Keywords Program features • Raw code • Abstract syntax tree • Variables • Pointer • Instruction • Basic block • Procedure • Control flow graph • Call graph • Object inheritance • Object dependence • API call • Data flow • Procedure dependence graph • System dependence graph

2.1 Syntactic Features

2.1.1 Raw Code

The raw code of the program can be analysed directly. For source code this is the textual stream, possibly normalized by removing comments and whitespace. For binaries, the raw code is the byte sequences (Fig. 2.1).

Definition 2.1 Let Σ be an alphabet of symbols. The raw code of program p is defined by the function r that evaluates to a string over the alphabet.

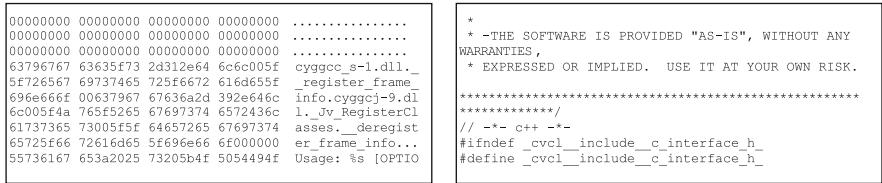
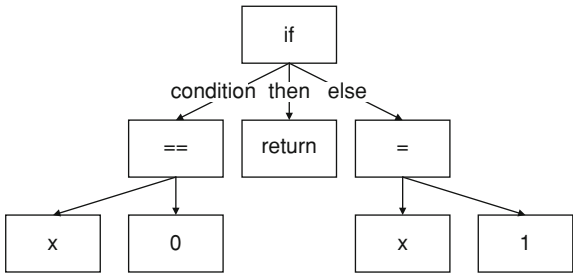


Fig. 2.1 Raw code for a binary (left) and source code (right)

Fig. 2.2 An abstract syntax tree (AST)



$$r : P \rightarrow S$$
$$p \rightarrow s, s \in \sum^*$$

2.1.2 Abstract Syntax Trees

Abstract syntax trees (AST) examine the syntax of source code and construct a tree representing the syntactical structure. For binaries, decompilation is required to reconstruct an abstract syntax tree (Fig. 2.2).

2.1.3 Variables

Variables represent the state of data. Programs typically maintain separate regions of memory for different classes of data handled by the run time environment. Run times may separate the stack from the heap to store data. The stack is used for local variables in a procedure and survives for the scope of that procedure or activation record. The run time creates a stack segment to achieve this outcome. In contrast, the heap is used for dynamically generated memory. Global variables conceptually belong to a different region than the heap, but for practical purposes are normally grouped together at run time in a data segment (Fig. 2.2).

Fig. 2.3 Typical pointer operations

<code>p = malloc</code>
<code>*p = q</code>
<code>p = *q</code>
<code>p = &q</code>
<code>p = q</code>

2.1.4 Pointers

Pointers are a type of variable that contain links or pointers to other variables. Pointers can be dereferenced, which allows for referencing the data the pointer is pointing to. Pointers may allow pointer arithmetic to be performed which allows for such operations as incrementing the value of a pointer. Some languages allow seemingly arbitrary pointer arithmetic, while other languages heavily restrict their use. Restricting pointer arithmetic allows for easier automated analysis (Fig. 2.3).

2.1.5 Instructions

Instructions capture the basic unit of computation. Computations can include such things as unary and binary operations, procedure or library calls. An instruction is defined by its operand and opcodes.

Definition 2.2 Let I be set of all instructions such that $I = \{(\text{opcode}, \text{operand}_1, \dots, \text{operand}_n)\}$

Definition 2.3 Let InstrSequence be a string of instructions such that $\text{InstrSequence} \in \sum^*$, $\sum = I$

2.1.5.1 Assembly

Assembly is a low level instruction format that can be executed on the native processing unit. It consists of opcodes which describe the type of operation to perform, and operands which are the arguments or parameters. Assembly language can be roughly divided into Complex Instruction Set Computing (CISC) architectures, or Reduced Instruction Set Architectures (RISC). RISC architectures favour simplified and small instruction sets while CISC architectures favour a rich and large instruction set. $\times 86$ is the dominant architecture for personal computing and is a CISC based architecture (Fig. 2.4).

2.1.5.2 Intermediate Representations

Instructions can be abstracted into intermediate representations. A common representation is Three-Address-Code which consists of three operands and one opcode. Typically, two fixed operands are inputs and the remaining operand is the

8d 4c 24 04	lea 0x4(%esp),%ecx	lea 0x4(%esp),%ecx
83 e4 f0	and \$0xffffffff,%esp	and \$0xffffffff,%esp
ff 71 fc	pushl -0x4(%ecx)	pushl -0x4(%ecx)
55	push %ebp	push %ebp
89 e5	mov %esp,%ebp	mov %esp,%ebp
51	push %ecx	push %ecx
83 ec 24	sub \$0x24,%esp	sub \$0x24,%esp
e8 6a 00 00 00	call 4011b0 <__main>	call 4011b0 <__main>
c7 45 f8 00 00 00 00	movl \$0x0,-0x8(%ebp)	movl \$0x0,-0x8(%ebp)
eb 10	jmp 40115f <_main+0x2f>	jmp 40115f <_main+0x2f>
c7 04 24 a0 20 40 00	movl \$0x4020a0, (%esp)	
e8 5d 00 00 00	call 4011b8 <_puts>	movl \$0x4020a0, (%esp)
83 45 f8 01	addl \$0x1,-0x8(%ebp)	call 4011b8 <_puts>
83 7d f8 09	cmpl \$0x9,-0x8(%ebp)	addl \$0x1,-0x8(%ebp)
7e ea	jle 40114f <_main+0x1f>	
83 c4 24	add \$0x24,%esp	
59	pop %ecx	
5d	pop %ebp	
8d 61 fc	lea -0x4(%ecx), %esp	
c3	ret	

Fig. 2.4 Assembly instructions and basic blocks

output. For unary operations, the extra operands are ignored. Using intermediate representation has the advantage of normalizing a complex instruction set into a series of simpler standardized operations.

Definition 2.4 Let $TAC = (opcode, operand_1, operand_2, operand_3)$

2.1.6 Basic Blocks

A basic block is a sequence of instructions that satisfy the following conditions:

- Execution flow can only enter the basic block through the first instruction.
- Execution flow can only exit the block at the last instruction.

A basic block can also be represented as a directed cyclic graph showing the data dependencies between instructions.

Definition 2.5 Let $InstrSequence(b)$ be a string of instructions such that $InstrSequence \in \sum^*$, $\sum = I$ for basic block b

2.1.7 Procedures

Procedures and functions are found in structured programming which allows for making modular maintainable code. A program uses a set of procedures $F = procedures(p) = \{f_1, \dots, f_n\}$.

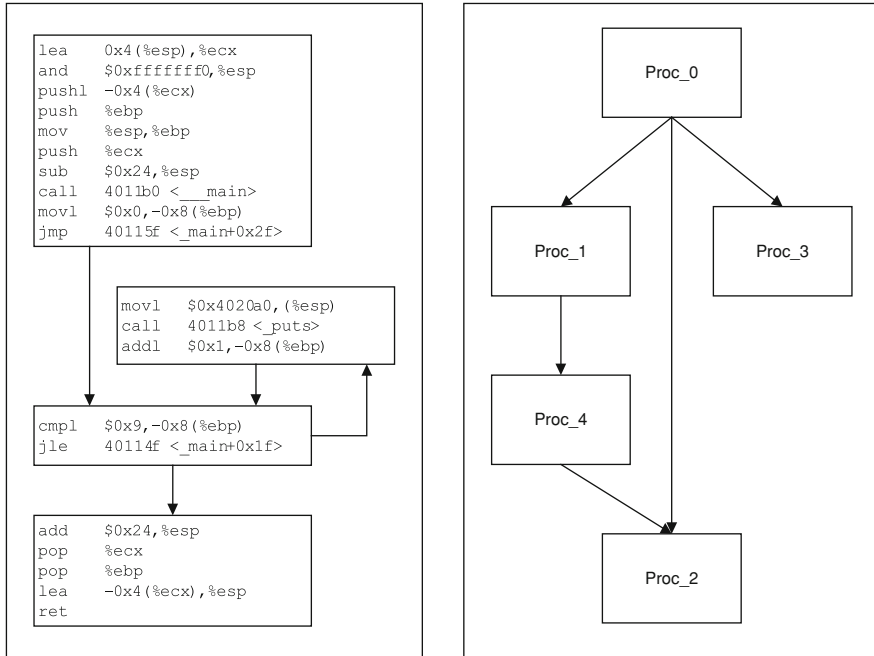


Fig. 2.5 A control flow graph (left) and a call graph (right)

2.1.8 Control Flow Graphs

The control flow graph is a directed graph representing the possible flow of execution within a procedure. The nodes in the graph represent basic blocks (Fig. 2.5).

Definition 2.6 The control flow graph of procedure f is the directed graph $C = (B, E)$ such that B is the set of basic blocks and E is the set of edges between them

Alternative representations of control flow are possible using graphs such as dominator trees or control dependency graphs.

Definition 2.7 $d \text{ dom } n$ or node d dominates a node n if every path from the start node to n must go through d

Definition 2.8 A node d strictly dominates a node n if d dominates n and d does not equal n

Definition 2.9 The immediate dominator or idom of a node n is the node that strictly dominates n but does not strictly dominate any other node that strictly dominates n

Definition 2.10 A dominator tree is a tree where each node's children are those nodes it immediately dominates

2.1.9 Call Graphs

The call graph represents the control flow between procedures and is again represented by a directed graph. If the program does not have recursive procedures, then the graph is acyclic. Like the control flow graph, dominator trees can be equally representative of the call graph (Fig. 2.5).

Definition 2.11 The call graph of a program is the directed graph $\text{Call-Graph} = (F, E)$ such that F is the set of procedures and E is the set of edges between them

The interprocedural control flow graph combines the control flow graphs with the call graph. It is defined as $\text{ICFG} = (B', E)$:

- The set of control flow graphs.
- Each control flow graph is given an additional exit node, which is successor to the set of return nodes in the cfg.
- For all basic blocks, a call instruction divides the block into two parts. The first part is connected to a call_return node, and that in turn is connected to the remaining basic block part.
- For each basic block that now ends with a call instruction, the block's successor is additionally the control flow graph of the call target. The successor of the exit node of the target control flow graph is additionally the call_return node.

2.1.10 Object Inheritances and Dependencies

Objects come from object oriented languages which group procedures (known as methods) and data into modular units. Objects are related to other objects via inheritance of their functionality.

2.2 Semantic Features

2.2.1 API Calls

API calls represent calls to libraries and other imports.

2.2.2 Data Flow

Data flow statically represents the data at run time entering and leaving each basic block. Many types of data flow analyses [1] are possible including reaching definitions, liveness, available expressions, and very busy expressions.

2.2.3 Procedure Dependence Graphs

The control dependencies and data dependencies of a procedure can be represented in a single graph using a procedure dependence graph [2].

2.2.4 System Dependence Graph

The system dependence graph combines the set of procedure dependency graphs of each procedure into a unified representation.

2.3 Taxonomy of Features in Program Binaries

Programs may begin as source code, but are typically compiled into a target binary for execution on the native platform or in another run time environment. The target binary is a container for all the information necessary for its execution in the target environment. This container is known as the object file format [3].

2.3.1 Object File Formats

Object File Formats contain five types of data:

- Headers.
- Object Code.
- Symbols.
- Debugging Information.
- Relocations.

Most modern object files also contain:

- Dynamic Linking Information.

2.3.2 Headers

The object file format is often described by a variety of headers. Headers may be used to define where the object code, symbols, debugging information, etc., is present in the binary.

2.3.3 Object Code

Object code contains the code and data of the program. For native executables the object code can consist of assembly or machine code. For object file formats such as Java class files, the object code contains byte code which is the instruction set architecture of the Java Virtual Machine.

2.3.4 Symbols

Parts of the code, data and binary may be associated with symbolic names. These associations are organized and stored in a Symbol Table.

2.3.5 Debugging Information

The binary may contain debugging information such as line numbers of source code associated with object code, or naming of information for different codes or data.

2.3.6 Relocations

If the binary has not been associated with a specific load address at compile time, the binary may need to be link edited at runtime. Relocations or fixups contain the necessary information to bind the object code to a specific load address.

2.3.7 Dynamic Linking Information

If the binary requires the use of external libraries, then the names of the required library functions must be present. Likewise, if the binary's functions are being exported as a library, then this information must also be present.

2.4 Case Studies

2.4.1 Portable Executable

The Portable Executable (PE) format [4] is the native object file format for the Windows family of operating systems. It is a modern file format which can contain


```
/bin/ls:      file format pei-i386
architecture: i386, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x00401000
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text          00019528  00401000  00401000  00000400  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE, DATA
  1 .data          00004be4  0041b000  0041b000  00019a00  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  2 .eh_frame      00000004  00420000  00420000  0001e600  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss           0000103c  00421000  00421000  00000000  2**3
                  ALLOC
  4 .idata         00000ddc  00423000  00423000  0001e800  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

Fig. 2.6 The output of objdump on a PE executable

all the information we have described in this section. It is identified by a series of magic bytes in its headers. Object code is defined in PE sections and an Import Address Table allows for dynamic linking.

2.4.2 Executable and Linking Format

The Executable and Linking Format [5] is the object file format in use on Linux and other operating systems. It replaced the previous a.out object file format in Linux. The a.out object file format did not natively support dynamic linking and ELF brought a much more modern format to Linux and enabled the transition to shared libraries using dynamic linking. An ELF binary is identified by a magic sequence in its header. There are three types of ELF object files (Fig. 2.6).

- Executable Objects.
- Relocatable Objects.
- Dynamic Objects.

Executable objects have been linked and bound to an address. Relocatable objects have not been bound to a load address and require linking. Dynamic objects have both a relocatable view and an executable view—shared libraries use this format.

Dynamic linking is slightly different to the PE format and uses a Global Offset Table (GOT) and a stub call to the runtime linker to resolve imports.

2.4.3 Java Class File

Java class files [6] contain object code in sections defined in the file's headers. The object code is in the instruction format for execution on the Java Virtual Machine. Like the previous object file format, a sequence of marker bytes (the magic bytes) in the header identifies the file format.

References

1. Aho AV, Sethi R, Ullman JD (1986) Compilers: principles, techniques, and tools. Addison-Wesley, Reading MA
2. Ferrante J, Ottenstein KJ, Warren JD (1987) The program dependence graph and its use in optimization. *ACM Trans Program Lang Syst (TOPLAS)* 9(3):319–349
3. Levine JR (2000) Linkers and loaders. Morgan Kaufmann Pub, Massachusetts
4. Pietrek M (2002) Inside windows-an in-depth look into the Win32 portable executable file format. *MSDN magazine*, pp 80–92
5. Standard TI (1995) Executable and linking format (ELF) specification version 1.2. In: TIS committee, May
6. Lindholm T, Yellin F (1999) Java virtual machine specification. Addison-Wesley Longman Publishing Co., Inc., Boston



<http://www.springer.com/978-1-4471-2908-0>

Software Similarity and Classification

Cesare, S.; Xiang, Y.

2012, XIV, 88 p. 26 illus., Softcover

ISBN: 978-1-4471-2908-0