Static Slicing of Binary Executables with DynInst

Tuğrul İnce
```c
int method = SET;
int number = 0;
int x = 1, y = 2;
if (method == SET) {
    number = 42;
    printf(“Just set the number to 42”);
}
else {
    x = y = 4;
    printf(“Not setting variable number”);
}
printf(“Final Value %d\n”, number);
```
Motivation

- Slicing is historically used for:
  - Debugging
  - Software Maintenance
  - Parallelization

- Generally on the source code
  - Binary executables
    - Moving dynamic analysis to static
      - Function pointers
    - Improve code generation
    - Identifying malicious code
    - Reverse-engineering viruses
    - Binary Proﬁlers
Slicing

- **Weiser’s original definition**
  - identifying all program code that can in any way affect the value of a given variable
  - This is now called “static backward slicing”

- **Static Forward Slicing**
  - Identifying all statements and control predicates dependent on the variable in the slicing criterion

- **Dynamic Slicing**
  - Identifying program code that *actually* changes the value of a given variable, determined at runtime.
How to Determine a Slice

- **Construct a Program Dependence Graph**
  - A Combination of Data Dependency Graph and Control Dependency Graph

- **Identify Data Dependency**
  
  1. a := 3
  2. b := a

  **b depends on a**

- **Identify Control Dependency**

  1. if a=true then
  2.     b := 1
  3. else
  4.     c := 0

  **Both assignments depend on if statement**
int main() {
    register int k=0;
    register int i=0;
    register int j=0;
    
    if(i==0) {
        k=1;
        j=5-k;
    }
    else {
        k=7;
        j=k-5;
    }
    i=k;
    printf("Printing i, j and k %d
", i, j, k);
    return 0;
}
movl $0x0,0xffffffff8(%ebp)
cmpl $0x0,0xffffffff8(%ebp)
jne 0x8048475 <main+49>

movl $0x1,0xffffffffc(%ebp)
mov $0x5,%eax
sub 0xffffffffc(%ebp),%eax
mov %eax,0xffffffff4(%ebp)
jmp 0x8048485 <main+65>

movl $0x7,0xffffffffc(%ebp)
mov 0xffffffffc(%ebp),%eax
sub $0x5,%eax
mov %eax,0xffffffff4(%ebp)
mov 0xffffffffc(%ebp),%eax
mov %eax,0xffffffff8(%ebp)
movl $0x0,0xfffffff8(%ebp)

cmpl $0x0,0xfffffff8(%ebp)

jne 0x8048475 <main+49>

movl $0x1,0xfffffffc(%ebp)

jmp 0x8048485 <main+65>

movl $0x7,0xfffffffc(%ebp)

mov 0xfffffffc(%ebp),%eax

mov %eax,0xfffffff8(%ebp)
Implementation

- **Static Analysis**
  - DynInst loads executable in stopped state

- **Building Data Dependency Graph**
  - For each instruction in a basic block, determine registers/variables that are read/written
    - Not so easy, large instruction set
  - When an instruction reads a register/variable, mark it as dependent on the one that recently modified that reg/var
A node $V$ is post-dominated by a node $W$ if every directed path from $V$ to Stop contains $W$.

An instruction $Y$ is control dependent on another instruction $X$ iff

- There exists a directed path $P$ from $X$ to $Y$ with another instruction $Z$ in $P$, post-dominated by $Y$
- $X$ is not post-dominated by $Y
Challenges

- **Indirect Jump Instructions**
  - Hard to create control flow graph
  - Very common in switch statements
    - Follows a pattern

- **Aliasing**
  - Currently not handled
  - Pointers
  - Treat all memory as a single object
    - Overly Conservative
    - EEL’s approach
On-demand Computation

- Generation of Data and Control Dependency Graph is costly, so is Slicing
- Since it is static, it is enough to compute these graphs only once
- Therefore, they are computed only on-demand and stored until the execution finishes
Annotation Framework

- Many analyses generate data while examining instructions/functions etc.
  - Generally costly operations
    - Store the result!
- New analysis means new variable(s) added to class definition
  - Error prone
  - API changes
  - Requires rebuild
Annotation Framework

- Create a unified Annotation Framework instead
- Use a well-defined interface for each object that needs to be annotated
- Has to be extensible
  - Add new annotation types at runtime
- Support for storing metadata along with data
Annotation Framework Example

- Requires development effort
- Not desirable
  - Error-prone
  - Tedious
Annotatable

createAnnotationType(String)
findAnnotationType(String)
createMetadata(String)
findMetadata(String)
insertAnnotation(AnnotationType, Annotation*)
findAnnotation(AnnotationType, Annotation*, int=0)

BPatch_Instruction
BPatch_Function
Annotation Framework

Annotation

void* value
getValue()

setValue(void*)

AnnotationWithSource

source
getSource()

AnnotationWithConfidence

certaintyValue
getConfidence()
Example

BPatch_function function = ... ;
AnnotationType type =
    function.createAnnotationType("Slice");
Graph* slicingGraph = ... ;
function.insertAnnotation(type,
    new Annotation(slicingGraph));
......
function.findAnnotation(type, fillMe);
Summary

- **Slicing**
  - Status
    - Intra-procedural Slicing implemented for x86 Linux and Solaris 2.9
    - Inter-procedural Slicing is on the way
  - Aliasing not supported yet

- **Annotation Framework**
  - Status: Designed, at implementation stage
  - Unifies the way objects are annotated
  - Slicing will be the first user