The Dyninst Binary Code Toolkits

Drew Bernat
Matthew LeGendre
Bill Williams

University of Wisconsin
http://www.paradyn.org

April 2008
Walking Difficult Call Stacks

- Functions can produce call frames that are difficult to walk through
  - Optimize away frame pointers
  - Non-standard frame pointers
  - Regions where frame pointer is not yet set up

- New features in StackwalkerAPI
  - Use debug information from binary
  - Use static analysis on binary
  - Use heuristics to dynamically search call graph
StackwalkerAPI

- Simple interface for collecting call stacks
  ```
  walker = new walker(pid);
  walker->walkStack(...);
  ```

- Callback interface for customization
  - Walks through types of stack frames
  - Identifies types of stack frames
  - Looks up symbol names
  - Accesses target process
Basic stack walking is essentially following a linked list.
Signal handlers and instrumentation tools may add non-standard frame layouts.
- Optimized functions may trash frame pointers
- Functions may not yet have created stack frames, or could change stack frames during execution.
Stacks without frame pointers

- What we have
  - The return address from the previous frame.
  - A pointer to the top of the frame

- What we want
  - The return address for this frame
Techniques

- **Debug Information**
  - DWARF, STABS, etc... tell how to walk through a stack frame

- **Static Analysis**
  - Analyze the binary to understand what function stack frames look like

- **Heuristic Stack Searching**
  - Search through the stack to find stack frames
Debug Information

- Given a code address and process state gives the location of the return address.
  - E.g. the return address is 40 bytes above the top of the stack
  - E.g. the return address is at %ebp + 4

- Potential Issues
  - Is occasionally wrong
  - Not present in all binaries
  - Requires reading from the binary

- Usable through SymtabAPI
Use static analysis to determine $\Delta$, the distance to the top of the stack frame, for each instruction.
Undefined Stacks

- May see unknown changes to the stack pointer.

```
Δ=4  lea 4(%esp),%ecx
Δ=??? and $0xffffffff0,%esp
Δ=??? pushl 0xfffffffffc(%ecx)
Δ=??? push %ecx
Δ=??? sub $20,%esp
...
```

- May have conflicting stack values

```
Δ=28  xorl %eax, %eax
Δ=24  pop %eax
Δ=24  jmp ...
Δ=28  push $4
Δ=32  push $8
Δ=36  call foo

Δ=??? pop %edx
Δ=??? pop %eax
Δ=??? add $20, %esp
Δ=??? ret
```
Other Issues

- Some functions may “clean” their parent’s stack:

  ```assembly
  func1:
  push %eax
  push %ebx
  call func2
  ret

  func2:
  push %ebp
  mov %esp, %ebp
  ...
  leave
  ret 8
  ```

- Non-returning function calls interfere with analysis:

  ```assembly
  func1:
  ...
  push %eax
  push %ecx
  call abort

  func2:
  push %ebp
  mov %esp, %ebp
  ```
## Static Analysis Results

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Functions</th>
<th>Functions with Frame Pointers</th>
<th>Functions w/ Undefined Stacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc 4.1.2</td>
<td>234,955</td>
<td>233,787 (99.5%)</td>
<td>644 (0.2%)</td>
</tr>
<tr>
<td>icc v10.0</td>
<td>45,173</td>
<td>18,921 (41.9%)</td>
<td>3,019 (6.7%)</td>
</tr>
</tbody>
</table>
Static Analysis Results

- Manually found all non-returning functions in icc compiled gdb.

<table>
<thead>
<tr>
<th>Recognize Non-Returning</th>
<th>Functions</th>
<th>Functions with Frame Pointers</th>
<th>Functions w/ Undefined Stacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>6,067</td>
<td>489 (8.06%)</td>
<td>1,007 (16.6%)</td>
</tr>
<tr>
<td>Yes</td>
<td>6,051</td>
<td>649 (10.37%)</td>
<td>10 (0.16%)</td>
</tr>
</tbody>
</table>
Implementation

- **Now**
  - DyninstAPI runs analysis, produces result file
  - Result file is fed into StackwalkerAPI

- **Goal**
  - StackwalkerAPI runs analysis when needed
### Heuristic Stack Searching

- Use heuristics to search the stack for frames

#### An Address Space

- 0x00: 0x080483f5: call foo
- 0x080483f9: ...
- 0x40000000: Heap
- 0x41000000: Heap End
- 0xbfe00000: Stack Top
- 0xc0000000: Stack Bottom

#### The Stack

- ... 0x000000482
- 0x080483f9
- 0xbfed6b30
- 0x4010a7f0
- 0x0000000c
- ...

- An address is likely the top of a frame if ...
  - ... it points to an instruction that follows a call
  - ... the following address points into the stack
Questions?
Dyninst and Static Rewriting

DyninstAPI

Process Control

Object Parser

Object Output

Code Parsing

Instrumentation

Mutatee Process

a.out
push %ebp
mov %esp, %ebp
sub $0x16, %esp

libc.so
push %eax
push $0x8
call foo

libm.so
fstl %eax
fmul %st, %st(1)
ret

The StackwalkerAPI
Dyninst and Static Rewriting

Dyninst API

- Process Control
- Object Parser
- Object Output
- Code Parsing
- Instrumentation

Rewritten Binary

Target Binary

.text
push %ebp
mov %esp, %ebp
push %5
call foo

.data
0x00 0x00 0xA7 0x6B 0x58 0x99 ...

inc counter
push %ebp
jmp ...
inc counter
call foo
jmp ...

The Stackwalker API
A Static Binary Rewriter

- Uses the same abstractions and interfaces as Dyninst

- Instrument and modify objects on disk
  - Instrument once, run many times
  - Run instrumented binaries on otherwise unsupported systems (e.g. BlueGene)

- Operates on unmodified binaries
  - No debug information required
  - No linker relocations required
  - No symbols required
Static Vs. Dynamic Rewriting

**Static Rewriting**
- ✓ Amortize parsing and instrumentation time
- ✓ Easier to port (no process control)
- ✓ Generate more efficient modified binaries

**Dynamic Rewriting**
- ✓ Insert and remove instrumentation at run time
- ✓ Execute instrumentation at a particular time (oneTimeCode)
- ✓ Tool can respond to run time events (shared library loads, exec, ...)

The StackwalkerAPI
The Binary Rewriter Interface

- **BPatch_addressSpace**
  - Instrumentation
  - Image functions

- **BPatch_binaryEdit**
  - Open files
  - Write files

- **Dynamic Rewriting**

- **Static Rewriting**
  - Common Functionality
BPatch_addressSpace

- Use BPatch_addressSpace for static and dynamic code instrumentation.

```c
if (use_bin_edit)
    addr_space = bpatch.openFile(...)
else
    addr_space = bpatch.attachProcess(...)

...

addr_space->getImage()->findFunction(...);
addr_space->insertSnippet(...);
addr_space->replaceFunction(...);
```
BPatch_binaryEdit

- Open a file and its libraries for rewriting
  - a.out
  - libc.so
  - libstdc++.so
  - libpthread.so
  - libm.so

- Open a single file for rewriting
  - libbar.so

- Add new libraries to an application
  - a.out
  - libinstr_helper.so
New Dyninst Requirements

- Need to write object files
  - Add new code
    - e.g., Add generated instrumentation code
  - Write changes to existing code.
    - e.g., Write trampoline jumps
  - Reference symbols in other libraries
    - e.g., Generate instrumentation that calls libc’s write from the a.out
  - Update headers

- Start with Dyninst’s existing instrumentation and parsing mechanisms.
Modifying the Binary

Elf Header contains:

- Meta-information about the object
- Pointers to the locations of important sections
<table>
<thead>
<tr>
<th>elf_hdr</th>
<th>Program Header contains:</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog_hdr</td>
<td>• Information on how to lay out the binary in memory</td>
</tr>
<tr>
<td>dynamic</td>
<td>• The related section header contains information on how the binary is laid out on disk.</td>
</tr>
<tr>
<td>code</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td></td>
</tr>
</tbody>
</table>
Modifying the Binary

Dynamic Section contains:

- How to resolve references to other libraries.

- Multiple sections involved:
  - Dynamic Symbol Table
  - Dynamic Strings Table
  - Relocation tables
  - Symbol Versioning info
### Modifying the Binary

- Add space for instrumentation and relocated functions to end of object.

<table>
<thead>
<tr>
<th>elf_hdr</th>
<th>prog_hdr</th>
<th>dynamic</th>
<th>code</th>
<th>data</th>
<th>dyninstInst</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The StackwalkerAPI
Modifying the Binary

- Need to modify prog_hdr with new section info.
- Grow prog_hdr by copying it elsewhere.
- Linux bug means prog_hdr must follow elf_hdr
Modifying the Binary

- Add trampolines and other Dyninst modifications by patching existing code.
Modifying the Binary

- Need to add to dynamic for external references made by instrumentation.
- Cannot grow dynamic, so copy to end of object.
### Modifying the Binary

- Left old copies of sections in place.
- Updated pointers in elf_hdr to refer to new section locations.
- Did not move code or data sections.

<table>
<thead>
<tr>
<th>elf_hdr'</th>
<th>prog_hdr'</th>
</tr>
</thead>
<tbody>
<tr>
<td>elf_hdr</td>
<td>prog_hdr</td>
</tr>
<tr>
<td>dynamic</td>
<td>code w/ patches</td>
</tr>
<tr>
<td>data</td>
<td>dyninstInst</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>elf_hdr'</th>
<th>prog_hdr'</th>
</tr>
</thead>
<tbody>
<tr>
<td>elf_hdr</td>
<td>prog_hdr</td>
</tr>
<tr>
<td>dynamic</td>
<td>code w/ patches</td>
</tr>
<tr>
<td>data</td>
<td>dyninstInst</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>elf_hdr'</th>
<th>prog_hdr'</th>
</tr>
</thead>
<tbody>
<tr>
<td>elf_hdr</td>
<td>prog_hdr</td>
</tr>
<tr>
<td>dynamic</td>
<td>dynamic'</td>
</tr>
</tbody>
</table>

- `elf_hdr`: Holds information about the entire program.
- `prog_hdr`: Holds information about the program header.
- `dynamic`: Holds dynamic link information.
- `code w/ patches`: Code with applied patches.
- `data`: Data sections.
- `dyninstInst`: Instructions for dynamic instrumentation.

The StackwalkerAPI
Current Status

- Beta of binary rewriter in Dyninst 5.2.
  - Static binaries
  - Dynamic objects (but not inter-library calls)
  - System V ELF platforms (Linux, BG/L, Solaris, ...)
    - x86, x86-64, PPC, IA-64, SPARC

- Coming Soon in Dyninst 6.0
  - Inter-library calls in dynamic objects
  - Adding new libraries to an object
Questions?
The Deconstruction of Dyninst: The InstructionAPI

Bill Williams
University of Wisconsin
The InstructionAPI Goal

Support analysis algorithms

Provide a model that is:

• Simple
• Portable
• Abstract
Instructions Are Complicated

- **Abstract instruction model**
  - Portable
  - Filters information
  - Matches expectations of analysis algorithms

- **Platform-specific decoder**
  - Non-portable
  - No abstraction
  - Can build analysis if you know platform details

- **Register Transfer Lists**
  - Portable
  - Anti-abstraction
  - Great for code generation
  - Wordy & awkward for analysis
How Do We Build a Good Model?

- **Make a good component**
  - Abstract, platform-independent interfaces
    - Abstract away unnecessary platform/encoding specifics
    - Allow clean access to platform specifics

- **Make it useful to customers**
  - Concise model of syntax
  - Solid base for semantics
  - Direct queries for important analytic properties

- **Focus on analysis**
  - Good models exist for code generation
  - Code generation & analysis produce different abstractions
Comparison With Existing Tools

- **VEX (Valgrind, RTL)**
  - Doesn’t provide interface for analysis queries
  - Represents semantics

- **XED (PIN, Platform-specific)**
  - Doesn’t provide interface for analysis queries
  - Preserves all IA32 platform details
  - Closed-source license

- Both of these are focused on code generation, not analysis
The InstructionAPI System

Machine language buffer

Instruction Decoder

Instruction object

Instruction object

Instruction object

Operation

Operands
Our Instruction Model

We summarize information from this tree:

- At the instruction level
- At the opcode level
- For each operand
- For the elements of each operand

mov eax -> [ebx * 4 + ecx]
Use Cases

- Register liveness
- Stack frame analysis
- Evaluation and update
Use Case: Register Liveness

- Building pre-liveness from post-liveness
  - Input: set of registers live post-instruction
  - Get registers read, written
  - Live\textsubscript{pre} = (Live\textsubscript{post} U \text{read}(i)) – \text{written}(i)

```cpp
Instruction insn;
set<RegisterAST::Ptr> killedRegs;
set<RegisterAST::Ptr> liveRegs;

insn.getRegsRead(liveRegs);
insn.getRegsWritten(killedRegs);
set_difference(liveRegs, killedRegs);
```
Use Case: Stack Frame Analysis

- Find instructions that write the stack pointer: `isUsed(r_ESP)`
- If push/pop, get size of what’s pushed: `getOperand(i).size()`
- If add/subtract, evaluate the operand that’s not the stack pointer: `getOperand(i).eval()`
- If we have a known change, record it; if not, fall to **UNKNOWN**
Use Case: Evaluation & Update

mov eax -> [ebx * 4 + ecx]

Instruction defines memory at unknown address

Outside analysis gives us values for ebx, ecx

```
set<RegisterAST::Ptr> regsUsed;
set<Expression::Ptr> addressesDefinedExprs;
map<RegisterID, long> machineState;

insn.getRegsRead(regsUsed);
insn.getAddressesWritten(addressesDefinedExprs);
machineState[e_ebx] = 20;
machineState[e_ecx] = 1000;
UpdateRegisterValues(regsUsed, machineState);
addressUsed = addressesDefinedExprs.begin().eval();
```
Current Status

- IA32/AMD64 completed
- Integration into Dyninst in progress
  - Stack analysis completed
  - Liveness completed
  - Parsing coming soon
- Manual available
Extensions and Future Work

- Provided by UW:
  - Additional platforms
    - IA64, Power, SPARC
  - Value-added libraries
    - Machine state abstraction

- Components we’d like:
  - Operation semantics
  - Code generation IR
  - Instruction parsers
Questions?
The Stackwalker API

User Process

Binary

Binary Parsing

Symbol Table Parser

PE
ELF
COFF

Instruction Decoder

IA32
AMD64
Power

Code Parser

Binary Patching

Instrumentation

Code Gen

Stack-walking

Debugger Interface

Win
Linux
AIX

AST

Paradyn
SymtabAPI

- Generate new binary files
  - Add and modify sections
- Dynamic address mapping
  - Memory addresses to file offsets
- Parse debug information
  - Line information
  - Local variables and their types