Data Collection and Instrumentation

a summary by Adam Fuchs
Introduction to Binary Editing

Topics to be discussed include:
- Language differences
- Binary editing domain
- Uses of binary editing
- Reasons to edit binaries
Code exists in many different languages during its lifetime, generally in the categories of:

- Source code
- Object code
- Linked code

Any of these can be read/written/edited, not just source code.
Typical Code Lifetime Path

Languages

Source Code (c, java, ...)

Compilation

Compiled Code (o, la, so, ...)

Linking

Linked Code (exe, a.out, ...)

Abstractions

Objects

Variables

Functions

Symbol Table

Code/Data Separation

Stack, Heap

Control Flow Graph

Abstract Syntax Tree
Binary Editing Domain

- Compiled vs. Linked
- Editing vs. Augmenting
- Runtime vs. Compile Time (and between)
- Machine Specific vs. Independent
Use Categories of Binary Editing

- Emulation
  - Becomes translation in the limit
  - Good for experimenting with new concepts

- Observation
  - Augment output
  - Advanced debugging

- Optimization
  - Add low level optimizations
  - Java's JIT compiler kinda does this
Example Applications

- We want to augment a library's functionality without access to source code
  - Print out or log more internal state information?
  - Add low-overhead debugging?
  - Add access control?
- Some programs take hours to compile (I think this was more true 10 years ago), but we want to make rapid minor changes to them
- Some programs show odd behavior after running for hours/days/months...
- We found the critical path during execution...
Tricky Issues

- We need to preserve (desired) semantics of the original program
  - Linked and optimized code very difficult to understand
  - Some code can be obfuscated on purpose (not really an HPC issue)
- Without source code, static analysis must be used to provide adequate context
- Calculated jumps are difficult to analyze statically
- Self-modifying code is even worse...
The Papers:

• Executable Editing Library (EEL)
  – C++ Library for binary editing

• Dyninst
  – Post-Compiler Program Manipulation Tool
Domain
- Operates on compiled and linked code
- Full editing capabilities (add and modify)
- Pre-execution modification only
- Machine independent editing support

Primary Uses
- Emulation
- Optimization
- Some Observation
  - (no runtime patching)
Abstractions

Five Primary Abstractions
• Executables
• Routines
• CFG
• Instructions
• Snippets
Executable Deconstruction

1. Read and process symbol table, removing symbols that refer to data
2. If there is no symbol table, then search the code for obvious subroutines
3. Analyze code to find not-so-obvious routines
4. Construct a CFG from distillation of the conservative set of routines from steps 1-3
Control Flow Graph (CFG)

- A tool modifies CFG
  - Delete instructions
  - Add snippets to blocks and edges
- EEL modifies calls, branches, and jumps to preserve flow semantics.
EEL Instructions

- Based on RISC instruction sets
- Higher level abstraction of standard set of machine instructions
- Categories are common to many instruction sets
- May lack precision in representing certain CISC instructions
Snippets

- Encapsulation of foreign code to be added to an executable
- Specifies instructions, register sets, and call-back function
int main(int argc, char* argv[]) {
    executable* exec = new executable(argv[1]);
    exec->read_contents();

    routine* r;
    FOREACH_ROUTINE (r, exec->routines())
    {
        instrument(r);

        while(!exec->hidden_routines()->is_empty())
        {
            r = exec->hidden_routines()->first();
            exec->hidden_routines()->remove(r);
            instrument(r);
            exec->routines()->add(r);
        }
    }

    addr x = exec->edited_addr(exec->start_address());
    exec->write_edited_executable(st_cat(argv[1],
                                             ".count"),
                                x);

    return (0);
}

void instrument(routine* r)
{
    static long num = 0;
    cfg* g = r->control_flow_graph();
    bb* b;
    FOREACH_BB(b, g->blocks())
    {
        if (1 < b->succ()->size())
        {
            edge* e;
            FOREACH_EDGE (e, b->succ())
            {
                e->add_code_along(incr_count(num));
                num += 1;
            }
        }
    }
    r->produce_edited_routine();
    r->delete_control_flow_graph();
}

mach_inst incr_count_code[] =
{
    #include "incr_count.bin"
};

long incr_count_offsets[] =
{
    #include "incr_count.oft"
};

class incr_count_snippet
    : public tagged_code_snippet
{
public:
    incr_count_snippet()
    :
        tagged_code_snippet(incr_count_code,
                             sizeof(incr_count_code),
                             NULL,
                             NULL,
                             incr_count_offsets,
                             sizeof(incr_count_offsets))
    {
    }

    code_snippet*
    incr_count(long counter_num)
    {
        assert(0 <= counter_num);

        tagged_code_snippet* snippet
            = new incr_count_snippet();
        addr counter_addr = COUNTER_START + counter_num * sizeof(long);
        SET_SETHI_HI(*snippet->find_inst(1),
                     counter_addr);
        SET_SETHI_LOW(*snippet->find_inst(2),
                      counter_addr);
        SET_SETHI_LOW(*snippet->find_inst(3),
                      counter_addr);
        return (snippet);
    }
System-Dependent EEL

- Uses bfd, or Binary File Descriptor, library
  - Reads and writes executable files in different formats
- spawn tool
  - Separates language from machine description
  - Allows for machine independent file coupled with description file
  - spawn generates machine specific code from the two files
Machine Desc.

- Used by spawn to map abstract low level code to machine-specific code.
**EEL Experience**

- Drastically reduces amount of source code
  - qpt went from 14,500 to 6,276 lines
- Hinders program size and performance
- Could improve with better C++ compilers

<table>
<thead>
<tr>
<th>Tool Version</th>
<th>Program Size (bytes) (text &amp; data)</th>
<th>Run Time (sec.) (user + system)</th>
</tr>
</thead>
<tbody>
<tr>
<td>qpt</td>
<td>246,784</td>
<td>4.4</td>
</tr>
<tr>
<td>qpt -O2</td>
<td>164,864</td>
<td>3.5</td>
</tr>
<tr>
<td>qpt2</td>
<td>950,240</td>
<td>19.0</td>
</tr>
<tr>
<td>qpt2 -O2</td>
<td>810,976</td>
<td>8.4</td>
</tr>
<tr>
<td>qpt2 -ND</td>
<td>868,320</td>
<td>18.0</td>
</tr>
<tr>
<td>qpt2 -O2 -ND</td>
<td>720,864</td>
<td>7.7</td>
</tr>
</tbody>
</table>
EEL Current Status:

- Evolved, and extinct?
**Dyninst**

**Domain**
- Operates on compiled and linked code
- Full editing capabilities (add and modify)
- Compile time and runtime modification
- Machine independent editing support

**Primary Use**
- Observation
- Emulation and Optimization possible, but not discussed in this paper
Abstractions

Mutator

Mutator App

API

Dyninst Code

Machine Dependent Code

ptrace/procfs

Points

Application

dowork(int a, b)
{
    int i;
    for (i=0; i<16; i++) {
        bar(a, i);
    }
}

Snippets

Run-time Library
Interface Classes

- **B Patch** – Single instance, represents API
- **B Patch\_thread** – One per executing thread of original program
- **B Patch\_image** – One per thread, represents running program
- **B Patch\_module** – Subdivision of image
- **B Patch\_function** – Subdivision of module
- **B Patch\_point** – Place to insert a snippet
- **B Patch\_type** – Interface to find/use types of the original program
- **B Patch\_snippet** – Foreign code abstraction
Snippet Interface

Snippets are written in an abstract, machine-independent language:

- `BPatch_variableExpr`
- `BPatch_arithExpr`
- `BPatch_boolExpr`
- `BPatch_gotoExpr`
Dyninst Implementation

- Use standard OS services to attach and manipulate running programs
  - ptrace, /proc filesystem, etc.
- Use a series of “trampolines” to splice in snippets without changing code size, semantics
Inserting Code

Figure 2: Inserting Code into a Running Program.
Code Example

1. Bpatch bpatch;
2. BPatch_thread *appThread->bpatch.createProcess(pathname, argv);
3. BPatch_image *appImage = appThread->getImage();
4. BPatch_Vector<BPatch_point*> *points =
   appImage->findProcedurePoint("InterestingProcedure", BPatch_entry);
5. BPatch_variableExpr *intCounter =
   appThread->malloc(*appImage->findType("int"));
6. BPatch_arithExpr addOne(BPatch_assign, *intCounter,
7.   BPatch_arithExpr(BPatch_plus, *intCounter, BPatch_constExpr(1)));
8. appThread->insertBlock(addOne, *points);
BPatch_function *openFunc = appImage->findFunction("open");

BPatch_Vector<BPatch_snippet *> openArgs;

B Patch _constExpr fileName(argv[3]);
openArgs.push_back(&fileName);
B Patch _constExpr fileFlags(O_RDONLY|O_CREAT);
openArgs.push_back(&fileFlags);
B Patch _constExpr fileMode(0666);
openArgs.push_back(&fileMode);

B Patch _funcCallExpr openCall(*openFunc, openArgs);

B Patch _variableExpr *fdVar =
  appThread->malloc(*appImage->findType("int"));

B Patch _arithExpr openExpr(BPatch_assign, *fdVar, openCall);
appThread->OneTimeCode(openExpr);
```c
1  BPatch_function *writeFunc = appImage->findFunction("write");
2  BPatch_Vector<BPatch_snippet *> writeArgs;
3  BPatch_paramExpr paramBuf(1);
4  BPatch_paramExpr paramNbyte(2);
5  writeArgs.push_back(fdVar);
6  writeArgs.push_back(&paramBuf);
7  writeArgs.push_back(&paramNbyte);
8  BPatch_funcCallExpr writeCall(*writeFunc, writeArgs);
9  BPatch_boolExpr compareFd(BPatch_eq, BPatch_paramExpr(0),
10                             BPatch_constExpr(1));
11  BPatch_ifExpr logStdout(compareFd, writeCall);
12  appThread->insertSnippet(logStdout, *points);
```
## Performance

<table>
<thead>
<tr>
<th>application</th>
<th># of operations</th>
<th>ops/sec</th>
<th>dyninst time (sec)</th>
<th>gdb time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress95</td>
<td>32,513</td>
<td>406,655.7</td>
<td>0.08</td>
<td>74.35</td>
</tr>
<tr>
<td>li (xilmatch)</td>
<td>110,209</td>
<td>43,607.7</td>
<td>2.53</td>
<td>221.04</td>
</tr>
<tr>
<td>li (compare)</td>
<td>4,475</td>
<td>640.2</td>
<td>6.99</td>
<td>16.39</td>
</tr>
<tr>
<td>li (binary)</td>
<td>401</td>
<td>19.4</td>
<td>20.69</td>
<td>21.62</td>
</tr>
</tbody>
</table>

- Up to 1000x faster than gdb for frequently evaluated conditional breakpoints
- About the same as gdb for infrequently evaluated conditional breakpoints
Current Status

Alive and kicking:

v5.1 was released in June

www.dyninst.org
Questions?