New Developments in the Dyninst and MRNet Toolkits

Bill Williams
Paradyn Project

9th Petascale Tools Workshop
Tahoe, CA
August 3, 2015
Dyninst 9.0 Overview

- **New features:**
  - Memory optimizations
  - Initial ARM64 support
  - Improved TLS support

- **Research areas:**
  - Improved parsing & dataflow analysis
  - Stack frame modification interface
  - SD-Dyninst integration

- Git-head is near final, official release coming soon
MRNet 5.0 Overview

- LIBI integration
- Verified ARM64 support
- Bug fixes
- Officially released 7/30/15
Symtab memory optimization

- Lazy demangling
- Lazy line information parsing
- Have observed ~75% reduction in Symtab overhead from these changes
- Tradeoff: higher CPU cost at initial startup
# Symtab optimization breakdown

<table>
<thead>
<tr>
<th>Area</th>
<th>Pre-opt. MB</th>
<th>Pre-opt %</th>
<th>Opt. MB</th>
<th>Opt. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line info indexes</td>
<td>1600</td>
<td>31%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Libdwarf leaks</td>
<td>950</td>
<td>18%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>String copies</td>
<td>300</td>
<td>6%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Demangled names</td>
<td>1000</td>
<td>19%</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Mangled names</td>
<td>240</td>
<td>5%</td>
<td>240</td>
<td>18%</td>
</tr>
<tr>
<td>Exception blocks</td>
<td>280</td>
<td>6%</td>
<td>280</td>
<td>21%</td>
</tr>
<tr>
<td>Symbol indexes</td>
<td>150</td>
<td>3%</td>
<td>150</td>
<td>11%</td>
</tr>
<tr>
<td>Other</td>
<td>670</td>
<td>13%</td>
<td>670</td>
<td>50%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5190</strong></td>
<td><strong>100%</strong></td>
<td><strong>1340</strong></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>

Results obtained from `openFile` and a request for line information at a single address.
ParseAPI memory optimization

- Blocks, functions, etc. stored in interval trees
- Can be overlapping
- Overlap is rare
- Two types of interval tree: fast and safe
  - Fast assumes non-overlapping intervals, $O(n)$ space
  - Safe assumes most/all intervals overlap, $O(n \log n)$ space
ParseAPI memory optimization

Non-overlapping (fast) set of intervals

<table>
<thead>
<tr>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800 - 0x808</td>
</tr>
<tr>
<td>0x808 - 0x811</td>
</tr>
<tr>
<td>0x811 - 0x830</td>
</tr>
<tr>
<td>0x900 - 0x905</td>
</tr>
<tr>
<td>0x905 - 0x90A</td>
</tr>
<tr>
<td>0x1100 - 0x121C</td>
</tr>
<tr>
<td>0x121F - 0x12A0</td>
</tr>
</tbody>
</table>

Overlapping (safe) set of intervals

<table>
<thead>
<tr>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x830 - 0x835</td>
</tr>
<tr>
<td>0x831 - 0x835</td>
</tr>
<tr>
<td>0x830 - 0x835</td>
</tr>
<tr>
<td>0x121C - 0x121F</td>
</tr>
<tr>
<td>0x121D - 0x121F</td>
</tr>
</tbody>
</table>
ARM64-enabled components

- **SymtabAPI**
  - Build system support
  - Generally smooth port

- **ProcControl**

- **Stackwalker**
ARM64-enabled components

- SymtabAPI
- **ProcControl**
  - Most functionality was easy
  - Kernel bug
  - Lack of ptrace backwards compatibility
- Stackwalker
ARM64-enabled components

- SymtabAPI
- Proccontrol
- **Stackwalker**
  - 3\textsuperscript{rd} party support works
  - 1\textsuperscript{st} party support coming later
ARM64-enabled components

- SymtabAPI
- Proccontrol
- **Stackwalker**
  - ARM stack layout is unusual
  - Calls don’t save RA to stack

<table>
<thead>
<tr>
<th>Normal stack</th>
<th>ARM stack</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Slot</strong></td>
<td><strong>Contents</strong></td>
</tr>
<tr>
<td>0</td>
<td>RA</td>
</tr>
<tr>
<td>1</td>
<td>FP</td>
</tr>
<tr>
<td>2…N</td>
<td>Locals</td>
</tr>
</tbody>
</table>
New thread local storage (TLS) features

- **ProcControl**: read & write TLS variables in a process

- **Dyninst**: trampguards moved to TLS
  - No hard limits on # of threads
  - Faster instrumentation in cases where trampguards are enabled
Instruction representation challenges

- **Maintain accurate map of bytes to opcodes**
  - Instruction sets grow & change rapidly
  - Syntax is easy, semantics are harder

- **Maintain accurate understanding of operands**
  - Register sets grow and change rapidly, too

- **Documentation is highly variable**
  - Good: standardized XML (ARM)
  - Medium: scrapeable HTML (PPC)
  - Bad: dead tree/PDF (Intel)
Jump table improvements

- Principled slicing-based approach
- Improves performance of instrumented binary
- Handles arbitrary number of table levels
Normal jump table

### Source-level construct

```c
switch(x) {
    case 0:
    case 2:
        // ...
        break;
    case 3:
        // ...
        break;
    default:
        // ...
}
```

### Table entries

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x405100</td>
<td>0x401102</td>
</tr>
<tr>
<td>0x405104</td>
<td>0x401F00</td>
</tr>
<tr>
<td>0x405108</td>
<td>0x401102</td>
</tr>
<tr>
<td>0x40510C</td>
<td>0x401107</td>
</tr>
</tbody>
</table>

### Binary implementation

```assembly
CMP %RAX, 0x03
JA 0x401F00
JMP *(0x405100+4*%RAX)
```
Two-level Jump table example

Source-level construct

```c
switch(x) {
    case 0:
        // ...
        break;
    case 29:
        // ...
        break;
    case 100:
        // ...
        break;
    case 169:
        // ...
        break;
    default:
        // ...
}
```

Binary implementation

First level table

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x416bd4</td>
<td>0x0</td>
</tr>
<tr>
<td>0x416bd5</td>
<td>0x4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x416c7c</td>
<td>0x4</td>
</tr>
<tr>
<td>0x416c7d</td>
<td>0x3</td>
</tr>
</tbody>
</table>

Second level table

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x416bc0</td>
<td>0x4156ac</td>
</tr>
<tr>
<td>0x416bc4</td>
<td>0x4157d0</td>
</tr>
<tr>
<td>0x416bc8</td>
<td>0x41596a</td>
</tr>
<tr>
<td>0x416bcc</td>
<td>0x41599e</td>
</tr>
<tr>
<td>0x416bd0</td>
<td>0x41677e</td>
</tr>
</tbody>
</table>
## Non-jump table example

<table>
<thead>
<tr>
<th>Source-level construct</th>
<th>Binary implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>switch(i % 8)</code></td>
<td></td>
</tr>
<tr>
<td>`{</td>
<td>AND 0x7,%EAX</td>
</tr>
<tr>
<td>case 0:</td>
<td>JE 0x80d93c8</td>
</tr>
<tr>
<td>x[i] -= y[i];</td>
<td>LEA 0x80d93c5+9*%EAX, %EAX</td>
</tr>
<tr>
<td>++i;</td>
<td>JMP %EAX</td>
</tr>
<tr>
<td>case 1:</td>
<td>80d93c8: //case 0</td>
</tr>
<tr>
<td>x[i] -= y[i];</td>
<td>mov (%esi),%eax</td>
</tr>
<tr>
<td>++i;</td>
<td>sbb (%edx),%eax</td>
</tr>
<tr>
<td>// …</td>
<td>mov %eax,(%edi)</td>
</tr>
<tr>
<td>case 7:</td>
<td>80d93ce: // case 1</td>
</tr>
<tr>
<td>x[i] -= y[i];</td>
<td>mov 0x4(%esi),%eax</td>
</tr>
<tr>
<td>++i;</td>
<td>sbb 0x4(%edx),%eax</td>
</tr>
<tr>
<td>// …</td>
<td>mov %eax,0x4(%edi)</td>
</tr>
<tr>
<td></td>
<td>80d9404: // case 7</td>
</tr>
<tr>
<td></td>
<td>mov 0x1c(%esi),%eax</td>
</tr>
<tr>
<td></td>
<td>sbb 0x1c(%edx),%eax</td>
</tr>
<tr>
<td></td>
<td>mov %eax,0x1c(%edi)</td>
</tr>
</tbody>
</table>
Jump table principles

- Tables are contiguous
- Tables depend on a single bounded input value
- Tables live in read-only data or code
Jump table results

- **Glibc:** ~30% decrease in uninstrumentable functions, 20% increase in parse overhead
- **Newly instrumentable libc functions include:**
  - `strncmp`
  - `strcmp`
  - `memcmp`
  - `memset`
- **Normal binaries:** ~5% increase in parse overhead, 7% decrease in uninstrumentable functions
Gap parsing improvements

- Machine learning based model updated for current compilers
- ...and finally integrated into Dyninst
- No longer need to apply compiler-specific models
## Gap parsing results

<table>
<thead>
<tr>
<th>Version</th>
<th>Platform</th>
<th>Avg. Precision</th>
<th>Avg. Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dyninst 8.2.1</td>
<td>64-bit x86</td>
<td>98.1%</td>
<td>37.4%</td>
</tr>
<tr>
<td>Dyninst 8.2.1</td>
<td>32-bit x86</td>
<td>95.6%</td>
<td>53.9%</td>
</tr>
<tr>
<td>Dyninst 9.0</td>
<td>64-bit x86</td>
<td>94.7%</td>
<td>83.2%</td>
</tr>
<tr>
<td>Dyninst 9.0</td>
<td>32-bit x86</td>
<td>97.1%</td>
<td>93.8%</td>
</tr>
</tbody>
</table>

Test binaries are from binutils, coreutils, and findutils, built with icc and gcc, at –O0 through –O3.
Stack frame modifications

- Can add, remove, swap, randomize space on stack
- Operates at function scope
- Mostly a security-oriented feature
- Important prerequisite: understand the stack frame with stack analysis
Stack analysis improvements

- Stack analysis: for each register, what stack location does it point to?
  - TOP: does not point to the stack
  - Numeric height: relative to SP at function entry
  - BOTTOM: may point to anywhere on the stack

- More instructions analyzed precisely
  - Added support for sign extend, zero extend, more general math (including more LEA math)
  - Improved stack modification from covering 30% of SPEC 2006 functions to 60% at -O2
SD-DynInst integration

- Maintain instrumentation capability through:
  - Dynamically generated code
  - Obfuscated control flow

- Designed for malware

- “Any sufficiently advanced optimizer is indistinguishable from malware”

- Can capture control flow through exception handlers
Slicing improvements

- Better handling of control flow cycles
  - Data flow around a cycle may involve different instructions on each iteration
  - Need to distinguish between visited instructions and visited assignments
- Many bug fixes, improving slice precision and accuracy
Range-based interfaces

○ **Lesson from Symtab optimizations: exposing containers is inflexible**
  ○ Whole container must exist, even if user wants one element
  ○ Hard to change types or relocate data

○ **Instead, prefer ranges**
  ○ Begin/end interfaces like STL containers
  ○ Typedefs for readability
  ○ Key to enabling, e.g., lazy demangling
LIBI

○ Single interface for launching processes
○ Does not replace RSH or XT launch frameworks, but augments them
○ Contact Dorian Arnold for details
MRNet ARM64 support

- MRNet now supports ARM64/Linux
- Full set of features should work
- Has not been tested at large scale
- Uneventful port
MRNet bugs fixed

- Build system fixes to support ARM
- Low port numbers (<10000) now work
- Better XPLAT_RSH_ARGS support
- Filter load failures are reported to front end
Ongoing and future work

- Windows binary rewriter
- Exception table rewriting
- Further memory and CPU improvements
- Completing ARM64 port
- New instruction foundation for x86