Zipr++: Exceptional Binary Rewriting

Jason Hiser, Anh Nguyen-Tuong, William Hawkins, Matthew McGill, Michele Co, Jack Davidson
University of Virginia
Motivation
Why do binary rewriters care about EH?

- Required by the x86-64 ABI
  - C++ supports exceptions, many programs use them
  - Rust, Ada exceptions essentially required
- More than just exception handling (EH)
  - Stack unwinding
  - pthread_exit
  - Profiling, debugging
- A “pin and win” strategy only goes so far
  - Limits efficiency (pinned calls)
  - Limits possible transforms (e.g., stack frame size)
  - Limits randomization (pinned return addresses)
Unwinding with ELF

- Common Info. Entry
- Shared
- 1-2 per ELF

- Function range

- Frame Description Entry
- Per function

- “Language specific” data area

- Personality is generic “unwind this frame” function

```
.cpp_personality:

func3:
    call string()
    ...
    call func1
    ...
    ret

func3_lp:
    call ~string()
    ...
```

```
.eh_frame

CIE 1
Personality:
Program:
    DP_1
    DP_2

FDE 1
FDE 2
FDE 3
CIE:
Start: &func3
Size: 120
Program:
    DP_3
    DP_4
    DP_5
LSDA:
```

DWARF Program
- Specifies how to unwind
Catching Exceptions

- Call Site Table
  - One entry per call
- Action Table
  - One entry per "catch"
- Type Table
  - Pointers to types "caught"
- "Language specific" data area
- Landing Pad
- Per function cleanup (call destructors) and rethrow if not caught

**.text .gcc_except_table**

- LSDA3
- CS tab len: 1
- CS table[0] :
  - CS Start: 
  - CS Size: 5
  - Landing pad:
  - Action:
    - Action table[0] :
      - Type :
      - Next: null
    - Type Table[0] :
      - Type :

**.text cpp_personality:**

- func3:
  - call string()
  - ... call func1
  - ... ret
  - func3_lp:
    - call ~string()
    - ...

**.rodata std::type_info for int**
EH in ELF

- Header for quick lookup via binary search
Modifying the EH info

- Modifications in place (patching)
  - Fields are variable length-encoded and PC-relative
  - Small change requires full table rewrite
  - Range-based → hard for per-instruction edits
  - Places lots of burden on transformation

Build IR instead!
Zipr++ Architecture
Modifying the EH info

- **Step 1: Deconstruct to IR**
  - Parse the EH info
  - Throw away all the encodings, ranges, etc.
  - Record essentials in the IR
    - The dwarf program with each instruction
    - Catch information for each call site
Modifying the EH info

- Step 2: Manipulate during transformation
  - C++ API to access EH IR
  - Most transforms can ignore completely

- Examples
  - Stack Layout Transform
    - Updates dwarf program to update location of return address relative to stack pointer
    - 1 C++ class, ~285 LOC
  - Stamp (xor) return addresses
    - Updates how to read the return address from the stack
    - 1 function, 72 LOC
Modifying the EH info

- **Step 3: Reconstruct**
  - Layout code (already done by Zipr)
  - Simple greedy scans code top to bottom
    - Extend current FDE or create new FDE as necessary
    - Re-use existing or create new CIE as necessary
    - Create an LSDA for each FDE if required
  - Generate/compress tables
    - Emit/generate assembly
Evaluation (Null Transform)

Performance Overhead

11% → 3%
Stack Padding

Normalized Execution Time

- **444.namd**
- **450.soplex**
- **453.povray**
- **471.omnetpp**
- **473.astar**
- **483.xalancbmk**
- **geo.mean**

- **SLX Optimized Code Layout**
- **SLX Optimized Code Layout with EH Frame Rewriting**
Related Work

- Static rewriters that require compiler support
  - ATOM, Diablo, etc.

- Other static rewriters without support EH
  - SecondWrite, UROBOROS, Ramblr, etc.

- Dynamic Rewriters
  - Strata, Pin, DynamoRIO
Summary

- Handling EH info is important
  - C++, Rust, Ada
  - Pthreads, debugging, profiling, performance
- Much EH information is about encoding or compression
  - Lesson Learned: Discard $\rightarrow$ simpler!
- Performance gains are real!