DBILL:
An Efficient and Retargetable Dynamic Binary Instrumentation Framework using LLVM Backend

Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu, Jan-Jan Wu, Wei-Chung Hsu (NTU), Pangfeng Liu (NTU), Pen-Chung Yew (UMN)
Outline

- Motivation
- Background
- Issues
- Architecture of DBILL
- Case Study
- Evaluation
- Limitation
- Conclusion
Outline

- Motivation
- Background
- Issues
- Architecture of DBILL
- Case Study
- Evaluation
- Limitation
- Conclusion
Dynamic Binary Instrumentation (DBI)

- Add extra code to a program at the level of machine code as it executes.
- Application
  - Bug Detection
  - Profile
  - Replay
  - Fault Injection
  - Watch Point
State-of-The-Art DBI

- PIN [PLDI ‘05]
- DynamoRIO [VEE ‘12]
- Valgrind [PLDI ‘07]

All of them are Same-ISA DBI.
ARM Executables are Everywhere

- Apple Store / Google Play (ART)
- Majority of ARM based systems are embedded devices and hard to develop DBI tools.

We need Cross-ISA DBI !!
Pros of Cross-ISA DBI

- The host system (e.g., x86 PC/desktop/server) has much more resources.
- The host machine often has greater computing power.
- The host ISA has a larger address space (e.g., 64bit vs. 32bit).
Cross-ISA Dynamic Binary Translation (DBT)

Uses QEMU and LLVM as building blocks.

Leverage HQEMU to Develop Cross-ISA DBI.
LLVM Instrumentation Tools

- Address Sanitizer [USENIX ‘12]
- Memory Sanitizer from Google
- Data Flow Sanitizer
- Thread Sanitizer [WBIA ‘09]
- Profiling Tools … etc

They are compile-time instrumentation tool.

We want to leverage these tools on binary.
Design Goal of our DBI framework

- Efficiency
- Retargetability
- Cross-ISA support
- Easy transformation from LLVM compile-time instrumentation tools to DBI based tools
Outline

- Motivation
- Background
- Issues
- Architecture of DBILL
- Case Study
- Evaluation
- Limitation
- Conclusion
LLVM Compile-time instrumentation Flow

Compiler FE 1

LLVM IR

Compiler FE N

LLVM IR

.bc

Native CodeGen

Target Binary

.o

Object File

Linker

Static-linked Library

Executable

.exe

LLVM Pass Manager

LLVM Analysis Passes

LLVM Transform Passes

LLVM Instrumentation Passes

compiler-rt
Address Sanitizer (ASan)

- Compile-time Address Sanity Checker
- Allocates a red-zone (unaccessible area) at the beginning and the end of each block returned by a memory allocation routine
- Red-zone are recorded as unaccessible
- Insert a piece of code before any memory access instruction to check the state of the memory bytes to be accessed
LLVM Compile-time instrumentation Flow

- Compiler FE 1
- Compiler FE N
- LLVM IR
- LLVM IR
- LLVM IR
- LLVM IR
- Native CodeGen
- Target Binary
- .bc
- .o
- Linker
- Object File
- Static-linked Library
- Executable
- .exe
- LLVM Pass Manager
- LLVM Analysis Passes
- LLVM Transform Passes
- LLVM Instrumentation Passes

Institute of Information Science, Academia Sinica
Outline

- Motivation
- Background
- **Issues**
- Architecture of DBILL
- Case Study
- Evaluation
- Limitation
- Conclusion
Issues in incorporating LLVM instrumentation tools

- **Annotation issue**
  - Distinguish guest binary IR and emulation IR.

- **Hijack issue**
  - How to intercept specific function calls.
Outline

- Motivation
- Background
- Issues
- Architecture of DBILL
- Case Study
- Evaluation
- Limitation
- Conclusion
Dissemble Stage

Guest Program Binary

Guest Program

Function Hijack Layer

Shared Lib

Syscall

Hypercall

System Call Handler & Guest Request Handler

Target FE

TCG-FE

LLVM Pass Manager

LLVM Instrumentation Passes

Annotation

Instrumented LLVM

Execution Engine
Execution Stage
Function Hijacking

- Guest Program Binary
- Function Hijack Layer
- Guest Program
- Shared Lib
- Syscall
- Hypercall
- System Call Handler & Guest Request Handler
- LLVM Pass Manager
- LLVM Instrumentation Passes
- Target FE
- TCG FE
- Execution Engine
- LLVM
- Instrumented LLVM
Outline

- Motivation
- Background
- Issues
- Architecture of DBILL
- Case Study
- Evaluation
- Limitation
- Conclusion
Translate a Memory Access Instruction

1. **Guest Instruction:**
   movl $0x0, (%eax)

2. **TCG IR:**
   mov_i32 tmp2, eax
   movi_i32 tmp0, $0x0
   qemu_st32 tmp0, tmp2, $0xffffffffffffffff

3. **LLVM IR:**
   %3 = load i32* %eax, !guest !0
   %4 = inttoptr i32 %3 to i32 addrspace(256)*, !guest !0
   # GVA -> HVA translation
   %5 = ptrtoint i32 addrspace(256)* %4 to i64
   %6 = add i64 %5, 0x7f8e00000000
   /*
   Check code (About 17 LLVM IR) instrumented by ASan */
   store volatile i32 0, i32 addrspace(256)* %4, !guest !0

4. **Instrumented LLVM IR:**
   %3 = load i32* %eax, !guest !0
   %4 = inttoptr i32 %3 to i32 addrspace(256)*, !guest !0
   store volatile i32 0, i32 addrspace(256)* %4, !guest !0
Control Flow of Hijacking malloc()
Control Flow of Hijacking free()
Outline

- Motivation
- Background
- Issues
- Architecture of DBILL
- Case Study
- Evaluation
- Limitation
- Conclusion
Environment

- i386 and ARM input executable are compiled by GCC 4.7.3 with –O3.
- x86-64 machine has 3.3 GHz Intel Core i7 and 12 GB memory. ARM machine has 1.2 GHz ARM cortex-A9 and 1 GB memory.
Comparison

- Valgrind 3.9.0 as the baseline
- DBILL with QEMU 1.3.0 and LLVM 3.2
Valgrind Memcheck vs. DBILL ASan

The chart compares the normalized execution time of various benchmarks using Valgrind Memcheck and DBILL ASan. The x-axis represents the benchmarks, and the y-axis shows the normalized execution time.


- Valgrind Memcheck is indicated with a light blue bar, while DBILL ASan is indicated with a dark blue bar.

- The execution times are normalized to show a clear comparison between the two tools for each benchmark.
Memory Access Counts

Valgrind Memcheck

DBIIL ASan

Memory Access Counts (Billion)

0 5000 10000 15000 20000 25000 30000 35000 40000 45000

Relation between Speed Up Factor and Memory Counts
Sources of Performance Gain

- Fast linear mapping to look up a shadow memory
  - Valgrind Memcheck uses a two level page table-like approach.
- Register promotion for architecture state mapping
- Helper function inlining
Advantage of Cross-ISA

Valgrind Memcheck on ARM vs. DBILL ASan on x86-64
Source Code Changes

- LLVM instrumentation tool
  - LLVM pass
  - compiler-rt
    - Tool dedicated part
    - Common part

<table>
<thead>
<tr>
<th></th>
<th>LLVM Pass</th>
<th>Tool Dedicated</th>
<th>Common</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASan</td>
<td>26/1120</td>
<td>40/4392</td>
<td>98/4858</td>
</tr>
<tr>
<td>MSan</td>
<td>80/2055</td>
<td>28/2269</td>
<td></td>
</tr>
</tbody>
</table>
Outline

- Motivation
- Background
- Issues
- Architecture of DBILL
- Case Study
- Evaluation
- Limitation
- Conclusion
Limitation

- Binary Instrumentation
  - Unable to leverage type-based optimizations
  - Unable to insert red-zone for global and stack variables

- Function Hijack of Static-linked Binary

- Floating Point Instruction
Outline

• Motivation
• Background
• Issues
• Architecture of DBILL
• Case Study
• Evaluation
• Limitation
• Conclusion
Conclusion

- Leverage HQEMU to develop an efficient, retargetable and cross-ISA dynamic binary instrumentation framework.
- We demonstrate that DBILL can easily transform LLVM instrumentation tools to DBI based tools.
- DBILL achieves an average speed-up of 1.74X for x86-based instrumentation on SPEC CPU2006 INT benchmarks, and an average speed-up of 8.66X for ARM-based instrumentation, compared with Valgrind.