Dynamic Binary Instrumentation Framework using LLVM Backend

{ Yi-Hong Lyu } Institute of Information Science, Academia Sinica

Backgrounds

Dynamic Binary Instrumentation (DBI) is adding extra code to a program at the level of machine code as it executes. It could be use to bug detection, profile, replay, fault injection and watch point. Today, most popular state-of-the-art DBI systems such as Pin, DynamoRIO and Valgrind target the same instruction set architecture (ISA) where the guest binary and the host binary are based on the same ISA.

Objectives of our DBI

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

Why cross-ISA support

Many popular applications on both Apple Store and Google Play contain ARM native code. However, majority of ARM based systems are embedded devices and hard to develop DBI tools.
On the other hand, building a cross-ISA DBI system which runs ARM executables on an x86 machine has multiple advantages:

- The host system has much more resources
- The host machine often has greater computing power
- The host machines ISA has a larger address space (e.g. 64bit vs. 32bit)

Therefore, it is very attractive to build cross-ISA program analysis tools to instrument ARM executables on x86 based systems.
PS. Instrument ARMv7 or earlier executables on ARMv8 is still considered as a cross-ISA scenario.

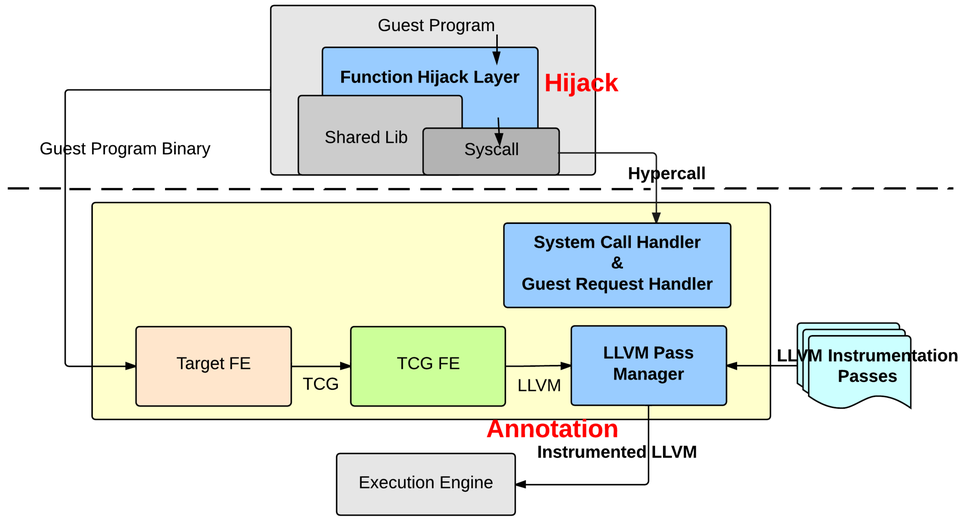
Why LLVM based DBI

There have been abundant compile-time instrumentation tools built upon LLVM. A LLVM based DBI system can quickly leverage these LLVM compile-time instrumentation tools.

Issues

- **Annotation issue** – Distinguish guest binary IR and emulation IR
- **Hijack issue** – How to intercept specific function calls

Architecture



Annotation

Guest Instructions:

movl $0x0, (%eax)

TCG IRs:

mov_i32 tmp2, eax
movi_i32 tmp0, $0x0
qemu_st32 tmp0, tmp2, $0xffffffffffffffff

LLVM IRs:

%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

Instrumented LLVM IRs:

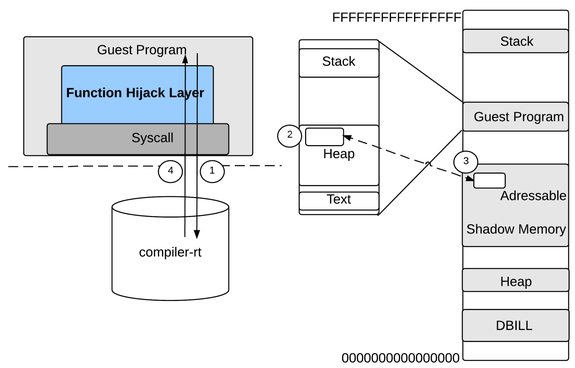
%3 = load i32* %eax,!guest !0
%4 = inttoptr i32 %3 to i32 addrspace(256)*,!guest !0
GVA -> HVA translation
%5 = ptrtoint i32 addrspace(256)* %4 to i64
%6 = add i64 %5,0x7f8e00000000
/* Check code (17 LLVM IR) instrumented by ASan */
store volatile i32 0,i32 addrspace(256)* %4,!guest !0

Hijack

Function Hijack Layer:

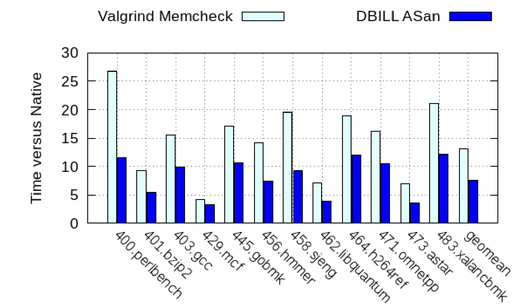
```
void *malloc(size_t size) {
  int ret = syscall (TARGET_NR_malloc, size);
  return (void *)ret;
}
```

The malloc function is compiled as a shared library and preloaded during guest binary loading time by the guest dynamic linker.



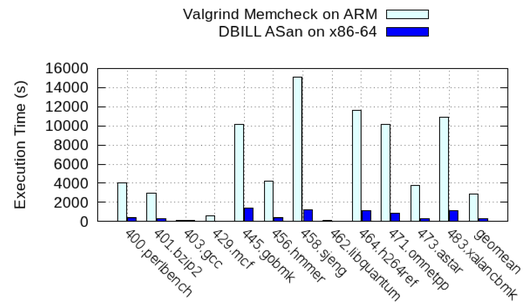
Same-ISA Results

Inputs are i386 executables. ISA of the host machine is x86-64.



Advantage of Cross-ISA

Inputs are ARM executables.



Source Code Changes

	LP	TD	C
ASan	26/1120	40/4392	98/4858
MSan	80/2055	28/2269	

Table 1: The table shows source code changes to incoporate Address Sanitizer (ASan) and Memory Sanitizer (MSan). LLVM instrumentation tools consist of two parts: LLVM pass (LP) part and compiler-rt part. Compiler-rt part consists of tool dedicated (TD) part (e.g. only for ASan or only for MSan) and common (C) part (shared by instrumentation tools).

Contact Information

LinkedIn www.linkedin.com/in/yihonglyu/
Email b95705030@ntu.edu.tw
Phone (+886) 963-233988