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

Yi-Hong Lyu  Ding-Yong Hong
Tai-Yi Wu  Jan-Jan Wu
Institute of Information Science,
Academia Sinica, Taiwan
{mitnick, dyhong, rogerwu, wuj}@iis.sinica.edu.tw

Wei-Chung Hsu  Pangfeng Liu
Dept. Computer Science & Information Engineering
National Taiwan University, Taiwan
{hsuwc, pangfeng}@csie.ntu.edu.tw

Pen-Chung Yew
Dept. Computer Science & Engineering,
University of Minnesota at Twin Cities
yew@cs.umn.edu

Abstract
Dynamic Binary Instrumentation (DBI) is a core technology for building debugging and profiling tools for application executables. Most state-of-the-art DBI systems have focused on the same instruction set architecture (ISA) where the guest binary and the host binary have the same ISA. It is uncommon to have a cross-ISA DBI system, such as a system that instruments ARM executables to run on x86 machines. We believe cross-ISA DBI systems are increasingly more important, since ARM executables could be more productively analyzed on x86 based machines such as commonly available PCs and servers.

In this paper, we present DBILL, a cross-ISA and re-targetable dynamic binary instrumentation framework that builds on both QEMU and LLVM. The DBILL framework enables LLVM-based static instrumentation tools to become DBI ready, and deployable to different target architectures. Using address sanitizer and memory sanitizer as implementation examples, we show DBILL is an efficient, versatile and easy to use cross-ISA retargetable DBI framework.

Categories and Subject Descriptors D.2.5 [Software Engineering]: Testing and Debugging; D.3.4 [Processors]: Runtime environments

General Terms Design, Performance

Keywords dynamic binary instrumentation framework, LLVM-based instrumentation, LLVM enhanced dynamic binary translation and instrumentation, memory bugs

1. Introduction
Dynamic Binary Instrumentation (DBI) is a core technique used for building debugging and profiling tools for application executables. DBI frameworks such as PIN [13], DynamoRIO [8] and Valgrind [14] have been widely used to build program analysis tools. Such tools include memory checkers, security violation detectors, profilers for code optimizations, and race detectors for assisting parallel programming.

Most state-of-the-art DBI systems target the same instruction set architecture (ISA) where the guest binary and the host binary are based on the same ISA. For example, an x86 executable is instrumented to run on x86 machines. It is uncommon to have a cross-ISA DBI system, where the guest and the host are based on different ISAs. For example, a system that instruments ARM executables to run on x86 machines is a cross-ISA instrumentation system.

We believe cross-ISA DBI systems will be increasingly more important because of the success of the Android platform for mobile devices, which uses ARM based executables. On the surface, Android applications are supposed to run under the Dalvik VM, and thus should be in the form of Dalvik code rather than ARM binaries. In reality, many popular applications include ARM native binaries in the package to maintain good runtime performance. Many popular applications on both Google Play and the Apple Store either contain ARM native code or are entirely ARM based. Although ARM based application executables are everywhere, program analysis tools for them are not easy to deploy since the majority of ARM based systems are embedded devices, which are difficult to develop on. On the other hand, x86 based PCs and servers are productive development machines and are commonly available. They also have much higher performance than ARM based systems. Therefore, it is very attractive to build cross-ISA program analysis tools to debug, profile, and analyze ARM executables on x86 based systems.
Building a cross-ISA DBI system which runs ARM executables on an x86 machine has multiple advantages: (1) the host system (i.e., x86 PC/desktop/server) has much more resources in terms of memory and storage for collecting traces, (2) the host machine often has greater computing power so analysis tools run faster, and (3) the host machines ISA has a larger address space (i.e., 64bit vs. 32bit) and thus memory mapping of the virtual machine is easier. Although the latest ARMv8 architecture supports 64bit virtual address space, its ISA is not compatible with earlier ARM architectures. So, exploiting the 64bit address space of ARMv8 for running dynamically instrumented ARMv7 or earlier executables is still considered as a cross-ISA instrumentation system.

To build a cross-ISA DBI framework, we could leverage a retargetable Dynamic Binary Translators (DBT) such as QEMU [6]. QEMU is a widely used system emulator for running ARM executables on x86 machines. The main contributor of QEMU’s high performance emulation is its DBT techniques. One drawback of QEMU is that its Tiny Code Generator (TCG) performs little code optimization. Recent research efforts [10, 11] have successfully combined LLVM [12] with QEMU, and with parallel code optimization and runtime optimization techniques [10], QEMU is not just a retargetable DBT framework, but a high performance, retargetable DBT framework. We intend to take advantage of such framework to build a cross-ISA DBI system.

Furthermore, because of the popularity of LLVM, there have been abundant static, compile-time instrumentation tools built upon LLVM, such as Address Sanitizer (detecting out-of-bound memory access) [18], Memory Sanitizer (detecting uninitialized read) [3], data flow sanitizer (for dynamic data flow analysis) [2], Thread Sanitizer (detecting race condition) [5], and profiling tools. This set of LLVM-based instrumentation tools are based on LLVM IR. A LLVM based DBI system can quickly leverage cross-ISA program analysis tools.

In this paper, using QEMU and LLVM as our building blocks, we propose a cross-ISA and retargetable dynamic binary instrumentation framework called DBILL (Dynamic Binary Instrumentation with LLVM). DBILL leverages both QEMU and the LLVM toolkit to enable LLVM-based instrumentations on binary code. DBILL leverages QEMU to translate guest binary code into TCG IR. It then translates TCG IR into LLVM IR to enable code optimizations and conduct instrumentation at the IR level. DBILL allows any instrumentation at the LLVM IR level and benefits from many existing static instrumentation tools based on LLVM IR.

The main contributions of this work are as follows.

1. Leveraging HQEMU [10], we have developed an efficient and retargetable dynamic binary instrumentation framework, DBILL, which supports cross-ISA, dynamic binary instrumentation.

2. We demonstrate that DBILL can easily transform LLVM IR based static instrumentation tools into DBI based tools. To the best of our knowledge, this is the first effort to successfully integrate LLVM-based instrumentation with a DBT system.

3. We demonstrate the effectiveness of DBILL in terms of performance and memory access counts using Address Sanitizer and Memory Sanitizer as implementation examples. 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 [14].

2. Background

Figure 1. Compile-time instrumentation flow in LLVM

Figure 1 depicts the compile-time instrumentation process in LLVM. First, the user program (written with a programming language) is translated to the LLVM intermediate representation (IR). Then, the LLVM IR is compiled to the target binary by the code generator. LLVM IR serves as a good medium for program analysis, transformation, and instrumentation. All these functions are governed by the LLVM Pass Manager. Finally, the linker links the generated object code with necessary libraries (e.g., compiler-rt) and produces the instrumented executables.

We use Address Sanitizer, an out-of-bound memory access checker developed by Google, as an example to illustrate two important steps in developing a compile-time instrumentation tool in LLVM. Address Sanitizer (ASan) is a red-zone-based memory checker [18] designed to detect addressable bugs. ASan inserts red-zones, which are unaccessible areas of a program, at the boundary of variables, and uses shadow memory to store the state of each address. Implementing ASan in LLVM requires the following two steps: (1) intercept memory allocation/deallocation library calls, and replace them with a routine for inserting red-zone and updating shadow memory, and (2) insert analysis code before each memory access to check if it will be out-of-bound.
Step (1) is implemented in compiler-rt as a modified standard library, and step (2) is implemented as an LLVM instrumentation pass, which instruments analysis code in the LLVM IR.

Existing LLVM-based instrumentation tools are mostly applied to LLVM IR derived from source programs. In the real world, many application programs and libraries are only available in binary form. By translating binary code to LLVM IR and incorporating LLVM-based instrumentation on the IR, a DBT system can leverage these LLVM-based instrumentation tools on binary code. Another benefit of such approach is that the shadow memory of the DBI system can be completely separated from the guest address space. We cannot achieve the same goal if we instrument program with LLVM static instrumentation tool first then run it on a DBT system.

3. System Architecture of DBILL

Incorporating an LLVM-based instrumentation tool in a DBT system would require the aforementioned two steps, but it raises two challenging issues. First, in a compile-time environment, function interception can be achieved simply by recompiling and relinking with the modified library in compiler-rt. In a DBT environment, however, recompiling and relinking is impossible, which makes function interception a challenging issue. Second, in a compile-time environment, the LLVM IR is all generated from the source code. Whereas, in a DBT environment, the LLVM IR includes those translated from the guest binary and others generated from the DBT itself for emulation purpose. An LLVM pass for instrumentation should only process those guest binary IR to ensure the correctness of the execution. A systematic annotation scheme to help an LLVM pass distinguish between guest binary IR and emulation IR is called for.

In this section, we present the system architecture of DBILL and the interaction between the system components. We also propose our solution to the two technical issues mentioned above. Finally, we present some optimizations to improve the performance of DBILL.

3.1 System Architecture of DBILL

Figure 2 shows the high-level architecture of DBILL. The input to DBILL is a guest program binary (e.g., executable, shared library or dynamic linker/loader). For a guest program binary, as indicated by the left path in Figure 2, the target front-end (FE) of QEMU emits code in the Tiny Code Generator (TCG) representation, which is translated by the TCG front-end. The TCG front-end performs a TCG to LLVM translation to emit code in LLVM IR representation. The LLVM Pass Manager inserts analysis code in the LLVM IR, and then a just-in-time (JIT) translator generates object code for the instrumented LLVM IR at runtime.

Some events, such as system calls and library function calls (e.g., memory allocation/deallocation routine calls), are pivotal in DBI. A DBI system needs to provide a mechanism to execute the corresponding callback functions when such events are triggered. Different DBI systems may use different mechanisms to handle system calls and library function calls. Our approach is adding two system components: the Function Hijack Layer and the Hypercall Handler (which is derived from the QEMU system call handler).

As indicated in the right path in Figure 2, the Function Hijack Layer intercepts the function calls to the shared library (e.g., memory allocation (malloc)/deallocation (free)), and then issues a hypercall, whose mission is to notify the DBI system when the event being monitored is triggered. When the DBI system receives a hypercall, it executes the corresponding hypercall Handler, which in turn executes the callback function registered for that particular event. For system calls, we adopt the approach of QEMU. When the guest program invokes a system call, QEMU executes the corresponding system call handler. If the system call is under the monitor of a DBI system/tool, then the system call handler will execute the callback function registered for that system call.

3.2 Interaction of System Components

In this subsection, we describe the interaction between the system components.

3.2.1 Disassemble Time - Target Front End and TCG Front End

The QEMU target front-end translates guest-binary executables into the TCG instruction set, called TCG IR. Then, the intermediate code is translated into target binary for execution. TCG IR is RISC-like, guest and host independent IR, and has only 142 opcodes in total. TCG IR can be classified into two categories: one that maps to guest executable instructions (e.g., ADD, SUB), and the other dedicated to DBT emulation (e.g., EXIT_TB). A DBI tool should only instrument codes in the TCG IR that belongs to the first category in order to accurately analyze the guest program itself.

We found that the TCG IR in the first category bears similarity to LLVM IR. This observation motivates us to use
QEMU target front-ends to preprocess guest binary codes (i.e., to translate them into TCG IR). We then implement a TCG front end to convert the TCG IR to LLVM IR. Such a two-phase conversion significantly simplifies the effort of translating guest binary into LLVM IR. Furthermore, since DBILL uses QEMU as its building block, it inherits all guest ISAs of QEMU, such as i386 (32-bit), x86-64 (64-bit), ARM (32-bit), PowerPC (32-bit), MicroBlaze (32-bit) and MIPS (32-bit).

3.2.2 LLVM IR annotation in DBILL
The TCG front end converts TCG IR to LLVM IR. Since an LLVM pass for instrumentation should only process those guest binary IR to ensure the correctness of the execution result, a systematic annotation scheme to help the LLVM pass distinguish between guest binary IR and emulation IR is called for. Our solution is as follows. In the TCG front end, we adopt the metadata scheme in LLVM to add annotation to the LLVM IR generated from guest binary. For example, we annotate a guest binary LLVM IR “%22 = add i32 %19, 4” with metadata “!guest”. The annotated IR will be “%22 = add i32 %19, 4, !guest !0”. In this way, an LLVM pass will be able to distinguish between guest binary IR and emulation IR with a simple check on the annotation. For example, the Address Sanitizer LLVM pass

```c
if (LoadInst *LI = dyn_cast<LoadInst>(I))
{
    //instrument analysis code
}
```

will perform instrumentation on the IR if the IR represents a load instruction (memory instruction). With a simple check, the modified Address Sanitizer LLVM pass as shown below will perform instrumentation on the load instruction only if the IR is annotated with "guest", meaning that it is generated from the guest binary.

```c
if (LoadInst *LI = dyn_cast<LoadInst>(I))
{
    if (LI->getMetadata("guest") && LI->isVolatile())
    {
        //instrument analysis code
    }
}
```

3.2.3 Instrumentation Time - Instrumentation Engine and LLVM Instrumentation Passes
The LLVM code block (LLVM IR) translated by the Target Front End and the TCG Front End is passed to the LLVM pass manager, which manages and processes all LLVM instrumentation passes. An LLVM instrumentation pass is a special kind of LLVM pass designed specifically for instrumentation purpose. The LLVM IR at this stage has been annotated as either guest binary IR or emulation IR. An LLVM instrumentation pass only analyzes and instruments codes to guest binary IR. Since the annotations are specific to DBILL, they have no effect on other LLVM passes.

3.2.4 Execution Time - Execution Engine
During execution, LLVM JIT (the Execution Engine) is used to invoke the appropriate code generator at runtime. LLVM JIT leverages the code generation ability of LLVM static compiler to generate high quality code, translating one basic block at a time. Since we use LLVM JIT in DBILL, the host ISA of DBILL is limited to the ISAs supported by LLVM JIT, including i386 (32-bit), x86-64 (64-bit), ARM (32-bit) and PowerPC (64-bit).

3.3 Optimizations for High-Performance Instrumentation

3.3.1 Helper Function Inlining
Most guest instructions are translated to TCG IR and then LLVM IR, and then the JIT compiler generates binary code from the LLVM IR. Some complicated instructions that are not supported by the TCG frontend can be implemented as helper functions. A helper function is implemented as a C function, and is compiled into a binary when compiling the QEMU source code. During emulation, a helper function is translated into a function call instruction. Such a function call incurs extra overhead, including arguments passing, adapting the arguments (the prolog), and retrieving the result (the epilog). The prolog/epilog increases memory accesses. Our approach to reduce memory accesses consists of the following three steps: (1) translate the helper function to LLVM IR using LLVM-GCC/Dragonegg, (2) inline the translated helper function IR in the LLVM IR translated from TCG FE, and (3) pass the LLVM IR translated from TCG FE to the LLVM JIT compiler.

3.3.2 Register Promotion for State Mapping
QEMU emulates the guest register file using host memory. Therefore, if a register is accessed twice in a basic block, host memory will also be accessed twice. We promote frequently accessed guest registers to host registers. At the beginning of the translated code, the guest register is loaded to a host register. Read/write operations to the guest register are all performed on the host register. At the end of the translated code, the value of the host register is stored to the host memory dedicated for emulating the guest register set. Such optimization can reduce memory accesses significantly.

3.3.3 Cross-ISA DBT
Like Valgrind, DBILL is a heavyweight DBI system. Current heavy weight DBI systems may suffer high overhead due to runtime recompilation of instrumentation code. We mitigate the performance problem by emulating the instrumented code on a faster machine through cross-ISA dynamic binary translation. As shown in Section 7, instrumentation on a machine with weak computing power (e.g., ARM) can
cause substantial slowdown. By instrumenting an ARM executable on a machine with strong computing power (e.g., x86-64) through cross-ISA DBT, we could achieve better performance.

4. Implementation Details

In this section, we elaborate on general implementation details of DBILL, including the start up stage of DBILL, shadow registers and shadow memory management for metadata storage, and the function hijack layer. These principles are applicable to other dynamic instrumentation tools.

4.1 Start Up

The start-up stage is the period between invoking DBILL from command line and the first guest program binary fetch for instrumentation. The goal of this stage is to load DBILL, reserve address space for the guest program and shadow memory (to be introduced in Section 4.2), and load the guest program into the same process (see Table 1).

DBILL itself is loaded by dynamic linker/loader (ld-linux.so under Linux) in host environment. When DBILL is loaded, it first initializes some sub-systems, such as Target FE, TCG FE, LLVM JIT and stored code cache. Second, DBILL reserves address space for the guest program and shadow memory. Third, DBILL begins to load a guest program. DBILL uses the loader code ported from the Linux kernel. The ported version has replaced the host address space allocation routine with target_mmap, which is used to allocate address space within a guest program’s address space.

Some earlier same-ISA DBI frameworks use the same dynamic linker/loader to load themselves and the guest program. This may harm robustness. Besides, many same-ISA DBI frameworks and their guest programs use the same shared library simultaneously. This may cause some potential complications and conflicts, such as reentrancy, side effect, idempotence and thread safety. However, the problems mentioned above do not occur in DBILL because of the following two reasons: (1) In a cross-ISA environment, the guest program binaries and the DBILL executable are incompatible and thus cannot be loaded with the same dynamic linker/loader. (2) In a cross-ISA environment, the guest program has been translated before execution, and thus DBILL and the guest program do not use the same shared library.

4.2 Shadow Registers and Shadow Memory

A DBI system usually provides a shadow mechanism for the DBI tools to store the metadata. The metadata related to guest registers are referred to as shadow registers, and those related to guest memory are referred to as shadow memory. QEMU uses separate parts of the host memory address space to simulate guest register and guest memory. Therefore, each guest register and guest memory location has a distinct address. This allows us to use the host virtual address of a guest state (register or memory) as the key to define the shadow address of the guest state. The case study presented in Section 5 describes the memory mapping for shadow registers and shadow memory in more details using Address Sanitizer as an example.

4.3 Function Hijack Layer

Function Hijack Layer is implemented as a shared library, which consists of the functions to be intercepted by the DBI system. We use the following example to explain the interception of the malloc function call:

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

First, the malloc function is compiled as a shared library. Then, the function hijack layer (shared library) is preloaded during guest binary loading time by the guest dynamic linker. After that, all malloc function calls from the guest program would be intercepted and replaced by our own version of malloc. Our malloc implementation uses the hyper call mechanism (syscall(TARGET_NR_malloc, size)) to notify DBILL that the guest program calls malloc. DBILL then accepts the notification and executes the corresponding handler.

5. Case Study: Incorporating Address Sanitizer

In this section, we describe the process of incorporating existing LLVM-based instrumentation tools in DBILL, using Address Sanitizer [18] as an example. Address Sanitizer (ASan) is a compile-time address sanity checker. ASan inserts red-zone at the boundary of variables. ASan then uses shadow memory to store the state of each address. To detect illegal memory access, it inserts a piece of code to check the shadow before any memory read/write.

As a compile-time tool, ASan can detect heap/stack/global variables out-of-bound access and heap use-after-free. Out-of-bound access checking relies on inserting red-zone at the boundary of variables. Since inserting red-zones will change the memory layout and will require recompilation of the instrumented code, it cannot be applied to global variables and stack variables at runtime. Hence, a DBI, such as Valgrind/Memcheck and Dr. Memory [7], can only detect heap out-of-bound access and heap use-after-free. DBILL is a runtime framework, and therefore, is subject to the same constraint as Valgrind/Memcheck and Dr. Memory. Other tools that do not change memory layout of the guest binary and thus do not require recompilation, such as data flow sanitizer, memory sanitizer, and thread sanitizer, can be fully incorporated in DBILL.
In the following subsections, we describe the process of porting Address Sanitizers heap out-of-bound access and heap use-after-free detection tools. Note that the guest we consider in this paper is 32-bit and the host is 64-bit.

## 5.1 Address Sanitizer

The state of a memory byte is either addressable or unaddressable. When a memory allocation routine (e.g., malloc, new) allocates memory, the memory state is changed from unaddressable to addressable. Calling a memory deallocation routine (e.g., free, delete) causes the deallocated memory to enter the unaddressable state. An access to memory that contains bytes that are currently in an unaddressable state causes memory violation.

To catch memory violations, Address Sanitizer [18] allocates a red-zone at the beginning and the end of each block returned by a memory allocation routine. The bytes in the red-zone are recorded as unaddressable. If a program accesses these bytes, Address Sanitizer signals a memory violation error. Address Sanitizer uses shadow memory to store the state of memory bytes. It also inserts a piece of code before any memory access instruction of an application to check the state of the memory bytes to be accessed by the application.

The implementation of Address Sanitizer consists of two parts: an LLVM pass named AddressSanitizer and library functions in the compiler-rt library. The former is a FunctionPass, which instruments the piece of code before memory access instructions at LLVM function level. The latter is a set of modified memory allocation/deallocation routines, which insert red-zones between the allocated memory and update the shadow memory during execution. AddressSanitizer redirects memory allocation/deallocation call of an executable to the corresponding function of compiler-rt during link-time.

## 5.2 Basic Structure

GNU libc uses mmap to allocate a block of memory and manages it by itself to implement memory allocation/deallocation routine. Dynamic memory management routine is almost an user level routine, which does not always trap into DBILL. However, this would be a problem for us because we need to update shadow memory. We use function hijack layer to replace the original dynamic memory management routine with our own, which would send a hypercall to DBILL when called. Then the hypercall handler uses the function in compiler-rt to update the shadow memory. Finally, the LLVM pass AddressSanitizer is added into the LLVM pass manager.

## 5.3 Memory Layout

Table 1 shows the memory layout when a guest program runs on DBILL. An entry in the right column represents the location of the memory segment in the host virtual address space. Note that, from host OS point of view, DBILL and the guest program are user level processes. The Guest Program segment is used to store the loaded guest program and data, and the Sadow segment is used to store the metadata on memory use by the guest program.

All same-ISA shadow-based instrumentation tools, either compile-time or runtime, suffer the potential threats that address space may be exhausted. This issue arises because existing shadow-based tools put the guest program and its shadow memory into the same address space. Despite the advance of shadow memory encoding algorithms, such problem would not be perfectly solved in the same-ISA systems. In contrast, DBILL separates address space for shadow memory from that for a guest program when the word size of guest ISA is shorter than that of host ISA. Under such scenario, the guest program could use address space as much as the guest ISA granted, which preserves address space transparency. As shown in Table 1, all these memory segments do not overlap. The isolation between each of them provides protection.

Note that providing address space transparency requires that the host has sufficient memory space to store shadow memory separately from that for the guest program. 32-bit on 32-bit does not provide such transparency because the host machine does not have sufficient space to store shadow memory separately. In this paper, we demonstrate that address space transparency can be achieved for DBI on 32-bit guest and 64-bit host. We believe that address space transparency is also possible for 64-bit guest and 64-bit host because most operating systems and application programs do not use the complete address. We will study the address space transparency issue for 64-bit on 64-bit in the future.

## 5.4 Translating a Memory Access Instruction

In this subsection, we will describe the process of instrumenting a memory access instruction. We first use pseudo code to explain the process. We then show the code segment translated by DBILL.

A Memory access translated by a compiler looks like the following:

```c
*address = ...; // or: ... = *address;
```

After instrumentation, a memory access would be the following[1]:

<table>
<thead>
<tr>
<th>Module</th>
<th>Host Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBILL</td>
<td>0x00000060000000 – 0x00000060ddd000</td>
</tr>
<tr>
<td>[heap]</td>
<td>0x0000006800ce000 – 0x00000068ce4000</td>
</tr>
<tr>
<td>Sadow</td>
<td>0x1ff1c0000000 – 0x1ff1dfffffff</td>
</tr>
<tr>
<td>Shared Lib</td>
<td>0x7f7f1e7f94000 – 0x7f7f1e80867000</td>
</tr>
<tr>
<td>Guest Program</td>
<td>0x7f7f8e0000000 – 0x7f7f8ef7000000</td>
</tr>
<tr>
<td>[stack]</td>
<td>0x7ffffb64e5000 – 0x7ffffb6507000</td>
</tr>
</tbody>
</table>

Table 1. Memory layout when a guest program runs on DBILL
We take one i386 memory access instruction to demonstrate how DBILL works. In the guest instruction, the eax register holds the address of one variable. The purpose of this guest instruction is to reset the variable, referenced by the address, as 0. Such assembly code is very common in variable initialization.

The guest instruction is translated into TCG IR by QEMU Target FE; meanwhile, the TCG FE of DBILL will know that the guest instruction is a guest memory store operation via qemu_st, a TCG OP, given by QEMU Target FE. Consequently, DBILL translates the TCG IR into LLVM store instruction (with attribute volatile and with metadata !guest !0).

With the !guest !0 metadata and volatile attributes of load/store LLVM instructions, the Address Sanitizer Pass will confirm the LLVM instructions as a memory access of the guest program and then insert check code in the LLVM IR. The %4 LLVM virtual register contains the guest virtual address (GVA). DBILL uses %5 and %6 LLVM instructions to calculate the host virtual address (HVA) via %4. Among the instrumented LLVM IRs, the number 0x7f8e00000000 represents the starting address position of GVA (e.g., the guest_base). HVA is computed as GVA + guest_base. HVA of (%eax) is used as a key for shadow memory as mentioned in Section 4.2.

**Guest Instruction:**
```assembly
movl $0x0, (%eax)
```

**TCG IR:**
```assembly
mov_i32 tmp2, eax
mov_i32 tmp0, $0x0
qemu_st32 tmp0, tmp2, $0xffffffffffffffff
```

**LLVM IR:**
```assembly
%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 IR:**
```assembly
%3 = load i32* %eax,!guest !0
%4 = inttoptr i32 %3 to i32 addrspace(256)*,!guest !0
# GVA -> HVA translation
```

5.5 Memory (De-)Allocation Routine Called by Guest Program

![Figure 3. Control flow of memory allocation and deallocation](image)

Steps (1a) to (1d) in Figure 3 show the interaction between DBILL components when the guest program calls memory allocation routine (e.g., malloc). (1a) The replacement memory allocation routine is invoked. We use syscall with a particular system call number SYS_MALLOC in the replacement to trap into DBILL, and invoke the compiler-rt routine incorporated from LLVM. The compiler-rt routine will perform the following: (1b) Allocate a block of memory for the guest program and its red-zone, (1c) mark the shadow of the memory region for the guest program as addressable,
and on the contrary, mark the shadow of its red-zone as unaddressable, and (1d) return the address to the guest program.

Since a guest program can only access within its address space, we use target_mmap instead of host mmap to reserve a block of memory within guest address space and manipulate it. When we want to return the address back to the guest program, we need to subtract the address by guest_base because of the address translation from host to guest.

Steps (2a) to (2d) in Figure 3 show the interaction between DBILL components when the guest program calls memory deallocation routine (e.g., free). The replacement process is similar. DBILL would perform the following three steps: (2a) Retrieve the address (argument) from the guest program, (2b) deallocate the block of memory and its red-zone, and (2c) mark the shadow of the memory region for guest program as unaddressable. When we want to get the address passed from the guest program, we must add the address by guest_base because of the address translation from guest to host.

6. Limitation of DBILL

6.1 Limitation of Binary Instrumentation

In general, binary instructions can be categorized into three categories: arithmetic, logic, control flow and loads/stores. Both TCG IR and LLVM IR follow this categorization, and instructions of an ISA/IR in one category are always translated into the same category of instructions of another ISA/IR. Therefore, DBILL loses no binary-level semantics in the process of translation and instrumentation.

However, the LLVM IR translated from guest binary usually contains less information than that derived from source programs. Since high-level type information is unavailable in many binary codes, the loss of type information in the LLVM IR translated from guest binary makes any DBI framework, e.g. DBILL, Valgrind, or DynamoRIO, unable to leverage type-based optimizations. Despite this limitation, there are still many optimizations applicable with only binary information, such as register promotion, inlining of helper functions (corresponding to some guest instructions), etc.

Another limitation is caused by the fact that binary information is insufficient to insert red-zone for global and stack variables. Since one cannot tell merely from a binary to which variable a location refers, it is also difficult to change the memory layout of a guest binary.

6.2 Function Hijack of Static-linked Binary

The current implementation of function hijack layer depends on the preload mechanism provided by the guest dynamic loader. The preload mechanism can work on any instruction set architecture without writing additional codes. Though the preload mechanism applies to only dynamic-linked libraries, this fact does not limit the practicality of DBILL, since dynamic libraries are widely used to save memory and storage space.

Indeed, function hijack layer can also be implemented for functions in a static-linked library, which keeps a symbol table. With the symbol table, DBILL can learn when a function in a static-linked library is called and hijack the function call in advance. DBILL works for (1) dynamical-link libraries and (2) static binaries which are not stripped.

6.3 Floating Point Instruction

Currently DBILL only supports integer instructions because TCG IR in QEMU lacks representation for floating point instructions. The TCG translator of QEMU v 0.15.0 and up does not emit floating point instructions of the host machine. Instead, all floating point instructions are simulated by integer instructions. Such simulation causes significant slowdown of the guest application. One possible solution is to emulate all floating point instructions via helper functions provided by QEMU. By using the LLVM compiler infrastructure, such helper functions can be inline and get floating point host instructions generated directly. Another alternative is to extend TCG IR to support floating point operations.

DBILL can be applied to floating point instructions once either one of the solutions is implemented.

7. Evaluation

In this section, we present the experimental results to evaluate the effectiveness of DBILL. Since DBILL is a heavy-weight DBI framework based on LLVM and LLVM IR, we compare with the widely used, state-of-the-art heavy-weight DBI system, Valgrind. Comparison of performance and memory access counts of these two systems are reported.

7.1 Experiment Setting

All the x86-based experiments were conducted on a x86-64 machine with a 3.3 GHz quad-core Intel Core i7 processor and 12 GB main memory. The operating system is 64-bit Gentoo Linux with kernel version 3.5.0. The ARM-based experiments were conducted on pandaboard with dual-core ARM cortex-A9 MPcore processors at 1.2GHz each and 1 GB main memory. The operating system on pandaboard is 32-bit Gentoo Linux with kernel version 3.4.48.

SPEC CPU2006 integer benchmark suite (CINT) is tested with reference inputs in the x86-based experiment. All benchmarks are compiled by GCC 4.7.3 and with -O3 optimization level. QEMU version 1.3.0, LLVM version 3.2 and Dragonegg version 3.2 were used as the building blocks of our DBILL framework. We chose Valgrind version 3.9.0 as our comparison counterpart. The latest officially released Valgrind version 3.8.1 fails to run some SPEC benchmarks on pandaboard, such as 445.gobmk, since it does not support some ARM instructions. Therefore, we use the newest subversion 3.9.0 (retrieved from Valgrind repository) in our ARM-based and x86-based experiments.
As mentioned in Section 6, QEMU v 0.15.0 and up does not emit floating point instructions of the host machine, and thus causes significant slowdown of floating-point based guest application. For this reason, we only use SPEC CINT benchmarks in our experiments.

For ARM-based experiments, the benchmarks are cross-compiled by GCC 4.7.3 and with -O3 optimization level and full hardware floating point convention. In the ARM-based experiment, the large memory consumption of CINT reference inputs will make some benchmarks either swap or corrupt at run time. Therefore, for ARM, we test CINT with train inputs.

Although both Valgrind and DBILL are IR based (re-targetable) heavy-weight DBI frameworks, Valgrind is designed for same-ISA environment. For i386 guest program, Valgrind translates i386 to i386. DBILL is a cross-ISA DBI framework. It translates i386 to x86-64. For fair comparison, we measured the SPEC CINT performance on i386 and x86-64bit with reference input. The geometric mean of the run time ratios (x86-64 over i386) is 0.97, with only 3% difference between the two architectures. Based on this observation, in the rest of our experiments, we will not take the individual discrepancy into consideration in x86 based performance comparison between Valgrind and DBILL.

7.2 Performance Comparison of Valgrind Memcheck and DBILL ASan

Valgrind’s Memcheck supports both memory addressable bug detection and uninitialized read detection. On the other hand, DBILL Address Sanitizer (ASan) only supports memory addressable bug detection (uninitialized read detection is supported by the Memory Sanitizer tool). For fair comparison, we turn off the following Valgrind Memcheck options, –leak-check=no –show-possibly-lost=no – leak-resolution=low –undef-errors=no, to disable uninitialized read detection. Therefore, in this setting, both Valgrind Memcheck and DBILL Address Sanitizer can detect heap out-of-bound access and use-after-free bugs.

Figure 4 compares the performance of Valgrind Memcheck and DBILL Address Sanitizer on x86, with SPEC CINT2006 (reference input) as the guest binary. The y-axis is the normalized execution time against the native run of the SPEC CINT2006 benchmarks compiled with gcc -O3 -m32. On average, DBILL ASan achieves speedup of 1.74X against Valgrind Memcheck. Furthermore, DBILL has larger speedup factors on perlbench, sjeng, libquantum and astar. The reason is that DBILL incurs much lower memory access counts than Valgrind on these four benchmarks. More details are presented in Section 7.3.

7.3 Comparison of Memory Access Counts of Valgrind Memcheck and DBILL ASan

We use the perf tool to monitor the memory access counts of Valgrind Memcheck and DBILL ASan in SPEC CINT. As Figure 5 shows, Valgrind Memcheck has more memory access counts than DBILL ASan. The geomean of the ratios (Valgrind Memcheck / DBILL ASan) of memory access counts is 2.71. The reasons are (1) Valgrind Memcheck uses page table-like approach to look up a shadow. It requires two memory accesses to look up one shadow. On the other hand, DBILL Address Sanitizer uses a linear mapping approach, which requires only one memory access for each shadow look-up. (2) DBILL implements special instructions via helper functions provided by QEMU. The LLVM infrastructure used by DBILL will inline all these helper functions. The result is that the number of prolog/epilog in basic blocks is reduced, which in turn also reduces the memory access counts. (3) DBILL also implements register promotion of architecture state mapping, which also helps reduce memory access counts.

Figure 6 shows the relation between the ratios of memory access counts (Valgrind counts divided by DBILL counts) and the speedup factors (Valgrind time divided by DBILL time). The lower line represents the speedup factors, and the
upper line represents the ratios of memory access counts. As can be seen in Figure 6, the curves of the two lines are quite similar. Higher ratio of memory access counts results in larger speed-up, while lower ratio results in smaller speed-up.

Figure 7 compares the performance of Valgrind Memcheck and DBILL Address Sanitizer on ARM. The ARM binary of the SPEC CINT benchmarks are given as the guest binary. Valgrind translates the ARM guest binary to ARM host binary of the same ISA. DBILL translates ARM guest binary to x86-64 host. In Figure 7, the left bars represent the execution time of Valgrind/Memcheck on ARM, and the right bars represent the execution time of DBILL ASan on x86-64. Figure 7 demonstrates the benefit of instrumenting ARM codes on a faster machine such as x86-64 by cross-ISA DBT. The speed-up factors (Valgrind time on ARM / DBILL time on x86-64) are significantly higher than on x86 machine (Figure 4, geometric mean of 1.74X) with geometric mean of 8.66X.

7.4 Time Breakdown

Figure 8 shows the time breakdown of DBILL ASan. The total execution time is divided into three parts: the translation time, the execution time in code cache, and QEMU emulation time. The translation time consists of the time from changing guest instructions into TCG and then being translated into LLVM IR and finally become instrumented LLVM IR. The Execution time in code cache is the time of running guest program. The QEMU emulation time is the time that the host system emulates the guest events, such as handling system call, finding the translated code, etc. Furthermore, 403.gcc has the highest proportion of translation time among these benchmarks. The reason is that QEMU generates large number of code blocks for 403.gcc; however, these code blocks are executed very few times.

7.5 Performance of DBILL Memory Sanitizer

Figure 9 illustrates the performance of DBILL Memory Sanitizer (DBILL MSan). The left bars represent DBILL MSan time, and the right bars represents the native run of the SPEC CINT2006 benchmarks on x86. The average slowdown of DBILL MSan is 5.7X.
7.6 Source Code Changes in Tools

<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>NSan</td>
<td>60/2855</td>
<td>28/2269</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Source code changes in tools. The number on the left in each column indicates the lines of source code changed, and the number on the right indicates the total lines of source code in each part of the LLVM instrumentation tool.

Our implementation effort to incorporate Address Sanitizer and Memory Sanitizer is reported in Table 2. All sanitizer tools consist of two parts: the LLVM pass part and the compiler-rt part. Note that compiler-rt consists of a set of tool dedicated parts (e.g., only for Address Sanitizer or only for Memory Sanitizer) and a common part (i.e., shared by all sanitizer tools).

Since currently LLVM does not provide standard interface support for developing LLVM based DBI tools, to incorporate an LLVM instrumentation tool in DBILL, minor changes to the LLVM pass and the compiler-rt are required.

For Address Sanitizer, 26 lines of code are modified in the LLVM pass, 40 lines in the compiler-rt tool dedicated part, and 98 lines in the compiler-rt common part. For Memory Sanitizer, 80 lines of code are modified in the LLVM pass and 28 lines in the compiler-rt tool dedicated part. Note that the compiler-rt common part is shared by all sanitizer tools, and hence, the modification only needs be performed once.

7.7 Summary of Results

We compare the performance of dynamic binary translation (i.e., without any instrumentation code) of Valgrind and DBILL on x86, and found that Valgrind has 3.9X slowdown on SPEC INT compared with the native run, and DBILL has 3.4X slowdown. DBILL achieves speedup of 1.15X against Valgrind. The performance gains of DBILL DBT are mainly contributed by helper function inlining (about 5.5%), register promotion (default on), and other LLVM optimizations. With instrumentation codes (DBILL ASan and Valgrind MemCheck), on average, DBILL ASan achieves speedup of 1.74X against Valgrind Memcheck (Figure 4). The main reason is that DBILL ASan incurs lower memory access counts than Valgrind Memcheck (Figure 5). As explained in Section 7.3, the lower memory access counts is a result of more efficient shadow memory management in Address Sanitizer.

In the case of instrumenting ARM code on x86-64 machine, the speed-up factors (Valgrind Memcheck time on ARM vs. DBILL ASan time on x86-64) are significantly higher than on x86 machine, with geometric mean of 8.66X (Figure 7). Most of the performance gain is from cross-ISA DBI because x86-64 PC runs much faster than the ARM (Pandaboard) platform we used in the experiment.

8. Related Work

There are several fundamental ways for a DBI framework to represent code and allow instrumentation: probe-, copy-and-annotate- (C&A) and disassemble-and-resynthesize- (D&R) based. The probe-based way works by replacing instructions in the original program with trampolines that branch to the instrumentation code at runtime. On the contrary, C&A and D&R are JIT-based, which uses a code cache to store the compiled result. The difference between C&A and D&R is whether there exists an abstract form (intermediate representation) of the instruction during instrumentation.

The advantage of Probe-based instrumentation is the speed. In contrast, it cannot provide fine granularity because some instruction is hard to replace and maintain the semantic simultaneously, especially for variable instruction length ISA (e.g., x86).

To support the fine granularity, we usually introduce JIT-based binary instrumentation. The compiling overhead of C&A is less than D&R. In the meantime C&A cannot introduce more compiler optimization technology. The most drawbacks of C&A compared to D&R is the ability to implement cross-ISA instrumentation.

Pin [13] and DynamoRIO [8] are the most famous DBI frameworks on windows. Pin has two approaches to dynamic instrumentations: probe-based and C&A-based (JIT-based). The probe-based approach is a method to insert probe, which is a jump instruction that redirects the flow of control to the replacement function. Probes can only be placed on function boundaries. Pin and DynamoRIO also support C&A-based dynamic instrumentations. They copy the instruction from the guest program, manipulating it and store into code cache. The added analysis code is usually interleaved with the original code without perturbing its effects. Neither probe- nor C&A-based dynamic instrumentation are easy to support cross-ISA DBI.

Valgrind [14] is the most popular DBI frameworks on Linux. The binary translators in Valgrind and DBILL use the D&R-based dynamic instrumentation. That is, both of them translate the guest program into an architecture-independent IR, and the instrumentation and compilation are performed on the IR. Although the binary translation technology of Valgrind bears similarity to DBILL, it does not support cross-ISA instrumentation.

MemTrace [16] is a framework for memory tracing for 32-bit guest programs. Memory tracing means that execution of additional code (aka. memlet) for every memory access of a program. Tool writer uses the API provided by MemTrace to set and check shadows values for every byte used in the guest program. MemTrace use libdetox [15], a C&A-based binary translator during execution. Because libdetox relies on translation table, the translation overhead would be low but the retarget effort would be more. Unlike traditional DBI framework, MemTrace does not provide shadow registers mechanism.
LIFT [17] is an information flow tracking (also referred to as taint analysis) system for IA-32 guest programs. Information flow tracking is the technique that labels the input data from unsafe source as taint data, propagates the labels of data through computation (any data derived from taint data is also tainted data), and detects inappropriate use of taint data. LIFT is built on top of StarDBT [19], which is a C&A-based dynamic binary translator targets translation from IA-32 into x86-64 at user level.

PIRATE [20] and DBILL use D&R-based binary translator. Both of them rely on QEMU to translate guest program into TCG first, then translate TCG into LLVM to further generate host code. Although our implementation paradigm is relevant, the problem we aim to solve are different. PIRATE is dedicated to providing an architecture-independent dynamic information flow tracking system. In contrast, DBILL provides a general architecture-independent DBI framework.

RevGen [9] and x86-to-LLVM DBT [4] use dyngen, a translation engine of QEMU used before version 0.10, to translates x86 binary to micro operation first, then inline LLVM IR precompiled from micro operation implemented as C function. In QEMU 0.10 and newer versions, dyngen has been replaced by TCG.

9. Conclusion

In this paper, we have proposed an efficient and retargetable dynamic binary instrumentation framework called DBILL, which can incorporate LLVM-based instrumentation tools. We demonstrate that LLVM IR, which has only been used for compile-time instrumentation previously, is also an effective choice for implementing dynamic runtime instrumentation. We have successfully incorporated a number of existing LLVM-based instrumentation tools in DBILL. In this paper, we have presented the process of incorporating Address Sanitizer in DBILL. We have also conducted extensive experiments with two tools, an address sanitizer and a memory sanitizer, to evaluate the effectiveness of DBILL. The experiment results show that DBILL ASan has better performance and less memory access overhead than Valgrind Memcheck.

Acknowledgments

We would like to express our gratitude to Todd Mytkowicz and Mathias Payer for their very helpful comments and suggestions to improve this paper. We also thank Chun-Chen Hsu and Yi-Shan Lu for helpful discussions. This work is supported in part by National Science Council of Taiwan under grant number NSC102-2221-E-001-034-MY3.

References


