Measurement: Instrumentation

Vojtěch Horký Peter Libič Petr Tůma

2010 – 2021

Contents

1 Overview **1**

2 Source Code Instrumentation **2**

3 Bytecode Instrumentation **3**

4 Machine Code Instrumentation **7**

5 Instrumentation Overwriting Code **9**

6 Instrumentation Translating Code **15**

1 Overview

Instrumentation

Purpose

Instrumentation inserts measurement code (probes) into well defined program locations to facilitate data collection.

Examples of instrumentation:

- Static instrumentation in source files
- Dynamic instrumentation in object model
- Static or dynamic instrumentation in bytecode
- Static or dynamic instrumentation in machine code
- Static preparation in source files that allows dynamic instrumentation in machine code

Instrumentation

Collected Information

Information that is interesting but cannot be measured directly:

- Precise program traces
- Program state snapshots
- Function parameter values
- Anything else the program can observe ...

Applications

Useful for many dynamic analyses:

- Test coverage
- Race detection

- Taint tracking
- Origin tracking
- Memory usage checks

2 Source Code Instrumentation

Source Code Instrumentation

Benefits

- Relatively easy to insert
  - Source code is made to be modified
  - Automated tools sometimes struggle
- Program state naturally available
- Locations relevant to code structure

Challenges

- Instrumentation is subject to compiler optimization:
  - Optimized together with surrounding code
  - Can impact surrounding optimizations
  - Can be subject to code motion
- Conditional instrumentation possibly tricky
- Requires source code and compilation

Source Code Instrumentation

Manual

Programmer inserts measurement code into locations of interest.

- Fine grained placement control
- Tedious and possibly error prone

Assisted

Tool inserts measurement code into specified locations.

- Tool guarantees systematic coverage
- Specification of locations limited

For tool examples think about logging frameworks or macro processors.

Source Code Instrumentation Problems

Configurability

We want to have simple way to turn instrumentation on and off.

- Preprocessing
- Conditional calls

Think what a modern runtime will do with the following:

```
log.debug ("App␣results␣are␣'" + results.toString () + "'.");
log.debug ("App␣results␣are␣'{}'.", results);
if (log.isDebugEnabled ()) {
  log.debug ("App␣results␣are␣'" + results.toString () + "'.");
}
```

Source Code Instrumentation Problems

Reliability
We need reliable association between instrumentation and application code.

Think what a modern compiler will do with the following:

```
int sqrt_counter = 0;
inline double counted_sqrt (double x) {
  sqrt_counter ++;
  return (sqrt (x));
}
```

This is the square root function disassembly:

```
sqrt:   pxor    xmm1,xmm1       // Set xmm1 to zero
        ucomisd xmm1,xmm0       // Compare argument to zero
        ja      fail            // Negative argument check
        sqrtsd  xmm0,xmm0       // Compute square root
        ret                     // Register passing
```

3 Bytecode Instrumentation

Bytecode Instrumentation

Benefits
- Still relatively easy to insert
  - Bytecode contains much metadata
  - Compilation produces predictable bytecode
- Program state still naturally available
- Can be done at runtime without sources

Challenges
- Requires tools
- Not all languages and environments have bytecode
- Locations in bytecode possibly different from code structure

For tool examples consider ASM or DiSL.
Also aspect oriented frameworks such as AspectJ.

ASM

Library for bytecode manipulation.
Main features of ASM are:
- Core API based on visitor design pattern
  - `ClassReader` to generate events from class file
  - `ClassWriter` to generate class file from events
  - Transformations implemented as event pipes
  - Adapters for predefined transformations
- Tree API for in memory class file representation
  - Can build representation from Core API events
  - Can generate Core API events from representation

`http://asm.ow2.io`

Aspect Oriented Programming

Idea
What if we could express independent concerns by separate code fragments ?

- Logging
- Transactions
- Authorization

concern Program feature that stands apart from other features.
join point Program location where concern code resides.
pointcut Specification of a set of join points.
advice Code inserted at pointcut.
weave Insert advice.

Obviously AOP can be used to insert measurement instrumentation.

Some pointcut specifications can introduce significant perturbation.

AspectJ

Aspect oriented programming framework for Java.

Main features of AspectJ are:

- Byte code instrumentation at compile time and load time
- Declarative language for defining instrumentation points
- Instrumenting code written in Java

http://www.eclipse.org/aspectj

Instrumentation with AspectJ

```
aspect Measurement {

    // Select all executions of methods of class Main.
    pointcut allMainMethods (): execute (* Main.* (..));

    // Attach an around advice that measures time.
    Object around (): allMainMethods () {
        long timeBefore = System.nanoTime ();
        Object result = proceed ();
        long timeAfter = System.nanoTime ();
        System.out.println (timeAfter - timeBefore);
        return (result);
    }
}
```

AspectJ Join Points

Join points are:

- call and execution of a method or a constructor,
- execution of an exception handler,
- execution of a static initializer,
- read or write access to a field,
- execution of an advice.

```
execution (int SomeClass.someMethod (int))
execution (String *.get*Name (..)
call (* AnotherClass.* (String))
call (*.new (long))
handler (RemoteException+)
get (int *.counter)
```

Join points can be further constrained by:

- presence of an annotation,
- location within a class or a method,

- actual type of the current object, called object, arguments,
- control flow selected by particular pointcut,
- boolean expression.

```
this (SomeClass)
target (AnotherClass)
args (int, int, int, int)
cflow (SomePointcut)
within (SomeClass)
```

AspectJ Pointcuts

Pointcuts combine specifications of join points using standard operators `&&`, `||`, `!`.

```
pointcut callToSomeClassFromMain ():
    within (Main) && target (SomeClass+);

pointcut nonRecursiveCallToSomeClass ():
    call (* SomeClass.* (..)) && !within (SomeClass)
```

Pointcuts can make accessible variables in their context:

- current object,
- target object,
- arguments.

```
pointcut callToSomeClass (SomeClass o):
    call (* SomeClass.* (..)) && target (o);

pointcut namingSomething (String name):
    call (void *.set*Name (String)) && args (name);
```

AspectJ Advice

Advice can be associated with join point:

- before the join point,
- after the join point
  - when it returns normally,
  - when it throws an exception,
- around the join point.

```
before SomePointcut ():
    Object [] arguments = thisJoinPoint.getArgs ();
    for (Object argument : arguments) {
      System.out.println (argument);
    }

Object around AnotherPointcut ():
    return (proceed ());
```

More AspectJ Features

Aspects can include declarations across types:

- declare new fields,
- declare new methods,
- declare new constructors,
- introduce new parents,
- introduce new interfaces.

```
private interface HasCounter {}
declare parents:
    (SomeClass || AnotherClass) implements HasCounter;
private long HasCounter.executionCount = 0;
before (HasCounter o):
    execution (* *.* (..)) && this (o) {
    o.executionCount ++;
}
```

Bytecode Instrumentation Problems

Consider

Inserted instrumentation executes in the same virtual machine as program.

Can this cause any problems ?

- Deadlock between analysis and application
- State corruption inside application code
- Arbitrary assumptions in virtual machine
- Bytecode verification failures
- Shared reference handlers
- Coverage approximation
- ...

[1]

Program State Corruption

Coverage

Analyses may require total code coverage:

- Memory allocation tracking (leaves important)
- Taint tracking (and any other data flow)
- Reliable race detection
- ...

What happens if we try to instrument every class of the application ?

This includes `Object`, `String`, `System` classes.
These will likely be used by instrumentation and analysis code too.
It is (too) easy to run into infinite recursion or state corruption problems.

Dynamic Bypass Pseudocode

```
static boolean instrumentationOnline = false;

// Instrumentation snippet
if (!instrumentationOnline) {
  instrumentationOnline = true;

  // Actual instrumentation here

  instrumentationOnline = false;
}
```

Simple with single-threaded programs, can be tricky with multiple threads.

DiSL

[1]Based on Kell et al.: The JVM is Not Observable … doi:10.1145/2414740.2414747

A bytecode instrumentation framework with emphasis on coverage.

- Language and framework for Java bytecode instrumentation
- Similar to aspects, but read only and with more control
- Allows to write instrumentation snippets directly in Java
- Instrumentation points are selected using annotations
- Dynamic bypass is provided automatically

`http://disl.ow2.org`

Example Instrumentation Snippet

```
@Before (
    marker = BodyMarker.class,
    scope = "TargetClass.print␣(boolean)",
    order = 8)
public static void precondition () {
    System.out.println ("Precondition␣!");
}
```

snippet Code inserted as instrumentation.
marker Specification of location to instrument.
scope Specification of classes to instrument.
guard Instrumentation condition.

DiSL Architecture

Instrumentation Server

Standalone server responsible for creating instrumented classes.

- Standalone to minimize perturbation
- Instrumentation using ASM
- Optimizations

Application Client

Java virtual machine executing the instrumented application.

- JVMTI agent to intercept class loading process
- Remote communication with instrumentation server
- Also executes whatever instrumentation code is inserted

More DiSL Examples

Look at the examples from the DiSL distribution. Use the `ant run` command to execute individual examples.
smoke A minimum example demonstrating the `before` and `after` advice on a method.
local An example demonstrating the use of a synthetic local variable.
scope Limiting instrumentation scope.
marker Implementing custom marker.
guard Using guards to restrict instrumentation.
static Using and implementing static context.
dynamic Using dynamic context.

4 Machine Code Instrumentation

Machine Code Instrumentation

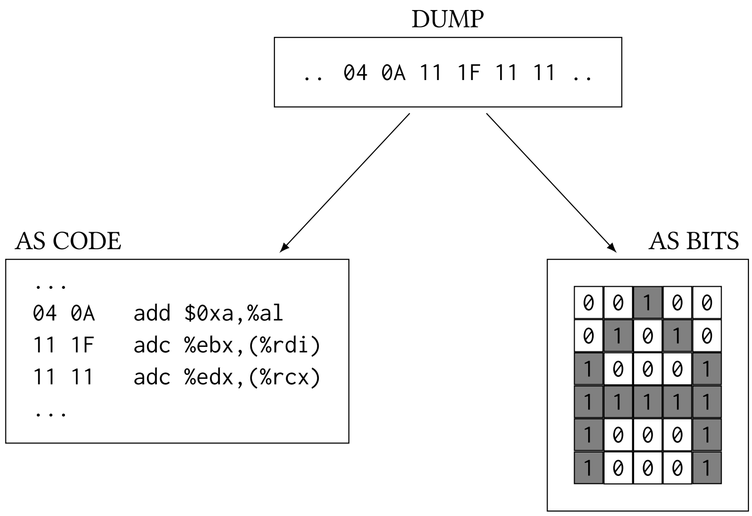
Benefits

- Machine code (almost) always available
- Looking at code in very fine resolution

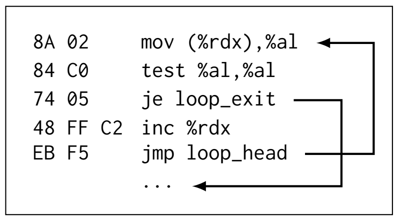
Challenges

- Machine code difficult to analyze
  - Mixing code and data
  - Variable length instructions
  - Very far from source code structure
- Inserting extra code difficult
  - No space for extra instructions
  - Register state must be preserved
- Some patterns complicate things
  - Auto generated or self modifying code
  - Computed branch targets

Recognizing Machine Code

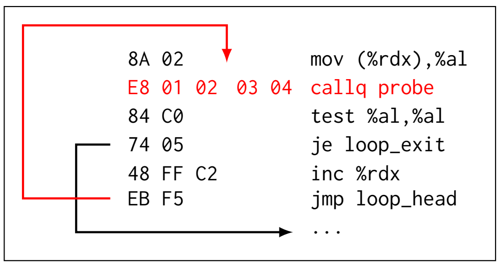


Shifting Machine Code



BEFORE PATCH

AFTER PATCH



Machine Code Instrumentation

Available tools use combinations of many techniques:

- Overwrite instrumentation locations
- Instrument only prepared locations
- Use dynamic translation

Example tools: DTrace, KProbes, PIN, Valgrind, DynamoRIO.

5 Instrumentation Overwriting Code

Overwriting Instructions

Instrumentation can be inserted by overwriting the instruction(s) at the target location.

Challenges

- Must overwrite only single location
  - As little as single byte on variable opcode length architectures
  - Single instruction on constant opcode length architectures
- Overwrite must be an atomic operation
- Original instruction must be replayed

- Specialized instructions
  - Intel `INT 3` opcode is single byte `0xCC`
  - Translates into `SIGTRAP` on Linux
  - But also writes on stack
- Hardware support
  - Explicit control transfers or barriers sometimes needed
- Trampolines

[2]

Instrument Prepared Locations

Instrumentation can be inserted at locations that were previously prepared for such use.

Challenges

- Identifying suitable locations
- Low overhead when not instrumented

- Prologues
  - Compilers can generate suitable function prologue
- Exported symbols
  - Reasonable location to expect instrumentation at
  - Often called through relatively standard code (PLT)
- Specialized instructions
  - Intel `NOP` has opcode variants for up to nine bytes
  - Atomic updates for long variants require some care

Linux Kernel Function Tracer

System for tracing calls to kernel functions:

- Instrumentation locations prepared by compiler `/sys/kernel/debug/tracing/available_filter_functions`
- Probes dynamically disabled and enabled
- Accessed through debug file system `/sys/kernel/debug/tracing/available_tracers` `/sys/kernel/debug/tracing/current_` `/sys/kernel/debug/tracing/trace`

[3]

Kernel Function Tracer Details

The kernel function tracer requires compiler support. When the kernel configuration includes the `FTRACE` infrastructure, the `-pg -mfentry` command line options are included among the compiler arguments. This option, originally intended for profiling, tells GCC to insert a call to a profiling probe at the start of each function. The kernel uses this call to produce function execution traces.
To minimize overhead, the kernel replaces all calls to the profiling probe with `NOP` instructions at boot time.

[2]See Chamith et al.: Living On The Edge … doi:10.1145/2908080.2908084
[3]See Linux kernel `Documentation/trace/ftrace.rst`

When tracing is enabled, probes are inserted in place of the NOP instructions.
To play with the function tracer and other tracing modules, try:

```
cd /sys/kernel/debug/tracing

# See what tracers are available.
cat available_tracers

# Enable the function tracer and see the output.
echo function > current_tracer
cat trace

# Constrain the function trace to only trace particular function.
echo do_mmap > set_ftrace_filter
cat trace

# Attach a trace filter command to particular function.
echo do_mmap:stacktrace > set_ftrace_filter
cat trace

# Enable the function graph tracer and see the output.
echo do_mmap > set_graph_function
echo function_graph > current_tracer
cat trace

# Pause and resume the current tracer.
echo 0 > tracing_on
echo 1 > tracing_on

# Disable the current tracer.
echo nop > current_tracer
```

Note that tracing is disabled while the trace file is being read. This makes sense but means the `tail` command does not show continuous tracing output.

The kernel function tracer can also be controlled through the `perf` tool.

To see how the function entry point is modified for tracing, install the kernel symbol information and disassemble the live kernel using the `crash` tool.
This is the output of `dis do_sys_open` with tracing disabled:

```
crash> dis do_sys_open
0xffffffff99315e00 <do_sys_open>:       nopl   0x0(%rax,%rax,1) [FTRACE NOP]
0xffffffff99315e05 <do_sys_open+5>:     push   %r15
0xffffffff99315e07 <do_sys_open+7>:     and    $0xfff,%cx
0xffffffff99315e0c <do_sys_open+12>:    push   %r14
0xffffffff99315e0e <do_sys_open+14>:    or     $0x8000,%cx
0xffffffff99315e13 <do_sys_open+19>:    push   %r13
0xffffffff99315e15 <do_sys_open+21>:    push   %r12
...
```

The same with tracing enabled:

```
crash> dis do_sys_open
0xffffffff99315e00 <do_sys_open>:       callq  0xffffffffc0ee1000
0xffffffff99315e05 <do_sys_open+5>:     push   %r15
0xffffffff99315e07 <do_sys_open+7>:     and    $0xfff,%cx
0xffffffff99315e0c <do_sys_open+12>:    push   %r14
0xffffffff99315e0e <do_sys_open+14>:    or     $0x8000,%cx
0xffffffff99315e13 <do_sys_open+19>:    push   %r13
0xffffffff99315e15 <do_sys_open+21>:    push   %r12
...
```

Remember to exit the tool between changes to see the updates.

Linux Static Kernel Trace Points

System for tracing predefined kernel events of interest:

- Inserted by kernel developers into appropriate locations `/sys/kernel/debug/tracing/available_events` `/sys/kernel/debug/` tree
- Probes dynamically disabled and enabled
- Accessed through debug file system `/sys/kernel/debug/tracing/set_event` `/sys/kernel/debug/tracing/trace`

4

Static Kernel Trace Points Details

Implementing a static kernel trace point entails defining the structure holding the trace record and implementing associated copy and print functions. See `https://elixir.bootlin.com/linux/latest/source/samples/trace_events`.
To play with the static kernel trace points, try:

```
cd /sys/kernel/debug/tracing

# See what trace points are available.
cat available_events

# Enable a trace point and see the output.
echo sched:sched_switch > set_event
cat trace

# See the structure of the trace record for a trace point.
cat events/sched/sched_switch/format

# Restricting a trace point with filter expressions.
echo 'prev_comm == "bash"' > events/sched/sched_switch/filter

# Associating a trigger with a trace point.
echo 'stacktrace:88' > events/sched/sched_switch/trigger
```

Each trace point is associated with an instance of the `tracepoint` structure, the trace point code is simply a function that tests the key field of the structure to see whether the associated trace point is enabled.

Linux Kernel Probes

Interfaces that allow to instrument:

- Any single instruction in kernel (Kprobes)
- Any function entry and exit in kernel (Return Probes)

5

Kprobes

Replace target instruction with breakpoint instruction

- Breakpoint instruction is single byte
- Control transfer similar to interrupt
- Jumps can also be used (faster)

Execute probe code

- Registers saved as in other interrupts
- Handler for both pre code and post code

Single step the replaced instruction

- Must be done inside interrupt handler

[4]See Linux kernel `Documentation/trace/tracepoints.rst`
[5]See Linux kernel `Documentation/trace/kprobes.rst`

– Requires understanding all instructions

Return Probes

Replace function entry with Kprobe

- Kprobe saves original return address
  - Limit on concurrently executing functions with return probes
  - Invocations exceeding limit are counted but not intercepted
- Kprobe modifies function return address to point to trampoline
- Trampoline is instrumented with another Kprobe at system boot time

Execute function code

- Function executes normally
- On return control is passed to trampoline Kprobe

Kernel Probe Details

See https://www.kernel.org/doc/html/latest/trace/kprobes.html for documentation and https://elixir.bootlin.com/linux/latest/source/samples/kprobes for sample.

Linux Static User Mode Probe Points

System for marking predefined user mode locations of interest:

- Inserted by application developers into appropriate locations

```
#include <sys/sdt.h>
STAP_PROBE (app, location)
STAP_PROBE1 (app, location, arg1)
STAP_PROBE2 (app, location, arg1, arg2)
...
```

- Probes compile into
  - A NOP instruction in the probe location
  - A probe record in the ELF stapsdt note section

The perf tool recognizes user mode probe points after scanning the binary with perf buildid-cache.
[6]

Linux User Mode Probes

System for tracing dynamically instrumented user mode code locations:

- Instrumentation code inserted by kernel on code load
- Controlled through debug file system /sys/kernel/debug/tracing/uprobe_events echo "<type> <file>:<offset> [arglist

Support included in the perf tool:

```
> perf probe -x <file> --funcs
> perf probe -x <file> --line <func>
> perf probe -x <file> --vars <func>:<line>
> perf probe -x <file> <func>:<line> [<var> ...]
```

[7]

Linux User Mode Probe Details

Use the perf tool with user mode probes. The listing uses the prime sieve example from https://github.com/d-iii-s/teaching-performance-evaluation/tree/master/src/experiment-prime-sieve.

[6] See man stapprobes
[7] See Linux kernel Documentation/trace/uprobetracer.rst

```
# List what functions are available for probing.
perf probe -x basic -F

# List what lines are available for probing within a function.
perf probe -x basic -L main

# List what variables are available for probing on a line within a function.
perf probe -x basic -V main:13

# Set a user mode probe on a line and include variable content in event record.
perf probe -x basic main:13 i primes
```

Once the probe is set, in can be used as any other trace point, for example with other `perf` commands. It can also be examined directly in the kernel tracing infrastructure.

```
# See how the probe was defined in the debug file system.
cat /sys/kernel/debug/tracing/uprobe_events

# Enable the probe tracing event and observe the trace and the profile.
echo 1 > /sys/kernel/debug/tracing/events/probe_basic/main_L13/enable
timeout 1 ./basic
cat /sys/kernel/debug/tracing/trace
cat /sys/kernel/debug/tracing/uprobe_profile
```

To see how the probe point is modified, keep the probe enabled and compare file disassembly with live code disassembly:

```
# File disassembly. Does not show the probe.
objdump --disassemble=main --source basic
```

```
...
        // Any number that is still marked
        // as potentially prime is prime.
        if (can_be_prime [i]) {
  4010cd: 80 ba 80 41 40 00 00  cmpb   $0x0,0x404180(%rdx)
  4010d4: 74 ea                 je     4010c0 <main+0x20>
...
```

```
# Live code disassembly. Shows the probe.
./basic & gdb -ex "disassemble␣/rs␣main" basic $!
```

```
...
21          // Any number that is still marked
22          // as potentially prime is prime.
23          if (can_be_prime [i]) {
   0x00000000004010cd <+45>:  cc              int3
   0x00000000004010ce <+46>:  ba 80 41 40 00  mov    $0x404180,%edx
   0x00000000004010d3 <+51>:  00 74 ea 8d     add    %dh,-0x73(%rdx,%rbp,8)
...
```

Note how the disassembly is confused by the fact that the seven byte `cmp` instruction is replaced with the one byte `int` instruction. The disassembler considers the leftover part of the `cmp` instruction opcode to be the opcode of the next instruction after `int`. This is what the processor would also do if the execution resumed immediately after the `int` instruction. The probe handler must therefore adjust the return address after executing the original `cmp` instruction. Essentially, this is done by single stepping the original instruction copied to a different location, details are available in kernel sources in `arch/x86/kernel/uprobes.c` (online `https://elixir.bootlin.com/linux/latest/source/arch/x86/kernel/uprobes.c`).

Linux Extended Berkeley Packet Filters

Framework for securely injecting code into kernel.

Code

Injected code written in limited bytecode:

- RISC style instructions
- Limit on instruction count (1M)
- Limit on branching (no loops)
- Static memory access checks

Bytecode supported by LLVM backend.

Maps

Data export through maps:

- In kernel key value stores
- Both global and per processor

[8]

More bpf Examples

Examine the bpftrace examples from https://github.com/iovisor/bpftrace.
It is also possible to examine the bpf programs produced by bpftrace. Consider the following example, taken from the bpftrace oneliners, which counts the number of system calls per process:

```
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
```

While the command is executing, we can locate and dump the bpf program:

```
bpftool prog list
```

```
...
96: tracepoint  name sys_enter  tag 0123456789abcdef  gpl
  loaded_at 2000-01-01T12:34:56+0000  uid 0
  xlated 216B  jited 125B  memlock 4096B  map_ids 33
...
```

```
bpftool prog dump xlated id 96
```

```
   0: (b7) r1 = 0                              ;zero used to initialize local variables
   1: (7b) *(u64 *)(r10 -24) = r1              ;current process name buffer
   2: (7b) *(u64 *)(r10 -16) = r1               ;of size 16 at frame-24
   3: (bf) r6 = r10
   4: (07) r6 += -24
   5: (bf) r1 = r6                             ;first argument is buffer address
   6: (b7) r2 = 16                             ;second argument is buffer size
   7: (85) call bpf_get_current_comm
   8: (18) r1 = map[id:33]                     ;first argument is map
  10: (bf) r2 = r6                             ;second argument is *key
  11: (85) call htab_percpu_map_lookup_elem
  12: (b7) r1 = 1                              ;initial count when no value
  13: (15) if r0 == 0x0 goto pc+2
  14: (79) r1 = *(u64 *)(r0 +0)
  15: (07) r1 += 1
  16: (7b) *(u64 *)(r10 -8) = r1               ;new count in local variable at frame-8
  17: (18) r1 = map[id:33]                     ;first argument is map
  19: (bf) r2 = r10                            ;second argument is *key
  20: (07) r2 += -24
  21: (bf) r3 = r10                            ;third argument is *value
  22: (07) r3 += -8
  23: (b7) r4 = 0                              ;fourth argument contains flags
  24: (85) call htab_percpu_map_update_elem
  25: (b7) r0 = 0
```

[8] See man bpf

```
26: (95) exit
```

The r10 register serves as a frame pointer to a 512 byte stack frame reserved for the function. Function arguments are passed in registers starting with r1, the return value is in r0.
The native version of the program is displayed with `bpftool prog dump jited id 96`.

6 Instrumentation Translating Code

Dynamic Instrumentation

Instrumentation can be inserted by translating code during execution.

Challenges

- Identifying code to translate
- Keeping execution overhead reasonably low
- Making translation invisible to application

- Code recognized during execution
  - Anything executed must be code
  - Translate in units of basic blocks
  - Chain basic blocks from hot paths into traces
  - Cache and reuse translations during execution
- Instrument inside basic block discovery notification
- Interface to internal code representation
  - Close to binary form in "Copy and Annotate" (PIN, DynamoRIO)
  - Close to compiler IR in "Disassemble and Resynthesize" (Valgrind)

PIN

Intel dynamic binary instrumentation tool. `http://software.intel.com/en-us/articles/pintool`

- C and C++ API
- Provides multiple instrumentation points such as Routine (RTN), Image (IMG), Instruction (INS)
- Instrumentation snippet is normal C/C++ code
- Operates in JIT mode or probe mode
- Supports x86

PIN Tool Example

```
int main (int argc, char * argv []) {
    if (PIN_Init (argc, argv)) exit (1);
    INS_AddInstrumentFunction (CountInstruction, 0);
    PIN_StartProgram ();
}

VOID CountInstruction (INS ins, VOID * v) {
    INS_InsertCall (ins, IPOINT_BEFORE, (AFUNPTR) DoCount, IARG_END);
}

VOID DoCount () { ... }
```

... run with `pin -t pintool.so -- command`

[9]

Valgrind

[9] Based on code from Intel PIN examples

Open source dynamic binary instrumentation tool. `http://valgrind.org`

- C API
- Compiler style intermediate representation (VEX)
- Instrumentation implemented as VEX manipulation
- Targets heavyweight instrumentation
- Supports x86, ARM, PPC, MIPS ...

Valgrind VEX Example

Original Instruction

```
add eax,ebx
```

```
------ IMark(0x123456, 2, 0) ------
# Connects VEX to original code address and length

t3 = GET:I32(8)   # Guest state offset 8 is EAX
t2 = GET:I32(20)  # Guest state offset 20 is EBX
t1 = Add32(t3,t2)
PUT(8) = t1

# Does not show flags and program counter updates
```

[10]

Valgrind VEX Example

Original Instruction

```
add [eax+4],edx
```

```
------ IMark(0x123456, 4, 0) ------
# Connects VEX to original code address and length

t3 = Add32(GET:I32(8),0x4:I32)  # Non flattened
t2 = LDle:I32(t3)               # Little endian load
t1 = GET:I32(16)                # Guest state offset 16 is EDX
t0 = Add32(t2,t1)
STle(t3) = t0                   # Little endian store

# Does not show flags and program counter updates
```

[11]

Debugging With valgrind

Valgrind debugging support can be used to examine the internal program representation on the fly. The example uses the prime sieve example from `https://github.com/d-iii-s/teaching-performance-evaluation/tree/master/src/experiment-prime-sieve`.

```
# Launch valgrind with debug server after 0 errors detected
valgrind --tool=lackey --vgdb-error=0 ./basic
```

```
==8086== Lackey, an example Valgrind tool
==8086== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote.
==8086== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==8086== Command: ./basic
```

[10] Based on code from Valgrind headers
[11] Based on code from Valgrind headers

```
==8086==
==8086== (action at startup) vgdb me ...
==8086==
==8086== TO DEBUG THIS PROCESS USING GDB: start GDB like this
==8086==   /path/to/gdb ./basic
==8086== and then give GDB the following command
==8086==   target remote | /usr/libexec/valgrind/../../bin/vgdb --pid=8086
==8086== --pid is optional if only one valgrind process is running
==8086==
```

```
# Launch gdb and connect to debug server in separate terminal
gdb ./basic
target remote | /usr/libexec/valgrind/../../bin/vgdb
# This should be after after primes counter increment
break basic.cc:26
cont
```

```
Breakpoint 1, main () at basic.cc:26
26              for (int j = 2 * i ; j < N ; j += i) {
```

```
disassemble/s main
```

```
...
18
19      // Check numbers from smallest to largest.
20      for (int i = 2 ; i < N ; i++) {
   0x00000000004010c0 <+32>:  add    $0x1,%rdx
   0x00000000004010c4 <+36>:  cmp    $0x8000000,%rdx
   0x00000000004010cb <+43>:  je     0x40110e <main()+110>
   ...
```

```
monitor v.translate 0x4010c0
```

```
==== SB 1234 (evchecks 0) [tid 0] 0x4010c0 main+32 basic+0x4010c0

------------------------ After instrumentation ------------------------

IRSB {
   t0:I64   t1:I64   t2:I64   t3:I64   t4:I64   t5:I64   t6:I1   t7:I64
   t8:I64   t9:I64   t10:I64   t11:I64   t12:I64   t13:I64   t14:I1   t15:I1

   DIRTY 1:I1 ::: add_one_SB_entered{0x58001090}()
   DIRTY 1:I1 ::: add_one_IRStmt{0x580010b0}()
   DIRTY 1:I1 ::: add_one_guest_instr{0x580010c0}()
   ------ IMark(0x4010C0, 4, 0) ------
   DIRTY 1:I1 ::: add_one_IRStmt{0x580010b0}()
   t2 = GET:I64(32)
   DIRTY 1:I1 ::: add_one_IRStmt{0x580010b0}()
   t0 = Add64(t2,0x1:I64)
   DIRTY 1:I1 ::: add_one_IRStmt{0x580010b0}()
   PUT(32) = t0
   DIRTY 1:I1 ::: add_one_IRStmt{0x580010b0}()
   DIRTY 1:I1 ::: add_one_guest_instr{0x580010c0}()
   ------ IMark(0x4010C4, 7, 0) ------
   ...
```

Note that the dump is shown in the valgrind terminal, not in the debugger terminal.