Observability
Observability

- **What is the system doing?**
  - Beyond the obvious (i.e. externally visible state changes)
    - Interactive debugging
    - Profiling
    - Tracing
    - Post-mortem analysis
  - Many flavors and requirements
    - Development / testing / production environments
    - Static / dynamic analysis
    - Intrusive / non-intrusive analysis
Instrumentation

- **Application of measuring instruments**
  - Recording of states and values
  - Safety
    - How the measurement affects the system under observation
  - Overhead
    - How the measurement affects the performance of the system under observation
      - While the instrumentation is active / inactive
  - Scope
    - Global vs. local phenomena
Interactive Debugging

- **Mechanisms**
  - Hardware debugging
    - JTAG serial interface to Test Access Ports
  - Breakpoints
    - Software breakpoints (BREAK, INT3)
    - Hardware breakpoints
      - E.g. DR0 to DR7 debug registers on x86
        - 4 linear addresses, trigger conditions (read, write, execute, I/O, area size), status
  - Single-stepping
    - E.g. trap flag in FLAGS + interrupt vector 1 on x86
  - Watchpoints
    - Hardware memory access breakpoints
    - Can be emulated via the paging mechanism
    - WatchLo, WatchH1 on MIPS
      - 1 physical address, trigger conditions (read, write)
Interactive Debugging

- **Debugger**
  - User interface
  - Exception handling code (privileged)
    - Kernel stub
      - Communicating with the debugger UI task
        - `ptrace(2), SIGTRAP, break-in thread (DebugActiveProcess())`
      - Remote debugging
        - Serial, FireWire, USB, virtualization extensions
      - Full-fledged in-kernel debugger (kmdb in Solaris, JDB in L4Re Microkernel)
      - 3rd party debugger (SoftICE, Rasta Ring 0)
      - Firmware debugger, hypervisor debugger stub
        - Non-maskable interrupts, SysRq

- **Debugging countermeasures**
Interactive Debugging in Linux

```c
pid_t pid = fork();
if (pid == 0) {
    ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    // Delivers SIGTRAP to the parent after successful exec
    // Automatically traces all signals
    execve(...);
}

int wstatus;
waitpid(pid, &wstatus, 0);
// Examine wstatus

// Configure which events are traced
ptrace(PTRACE_SETOPTIONS, pid, NULL,
      PTRACE_O_EXITKILL | PTRACE_O_TRACECLONE |
      PTRACE_O_TRACEEXEC | PTRACE_O_TRACEEXIT |
      PTRACE_O_TRACEFORK | ...);

// Examine and control the child
ptrace(PTRACE_GETSIGINFO, pid, NULL, siginfo);
ptrace(PTRACE_GETREGSET, pid, NT_PRSTATUS, iovec);
ptrace(PTRACE_PEEKTEXT, pid, remote_addr, local_addr);
ptrace(PTRACE_POKETEXT, pid, remote_addr, local_addr);
ptrace(PTRACE_SYSCALL, pid);
ptrace(PTRACE_CONT, pid, NULL, NULL);
```
Interactive Debugging in HelenOS

```c
errno_t rc;
async_sess_t *session = async_connect_kbox(task_id, &rc);
udebug_begin(session);

// Configure which events are traced
udebug_set_evmask(session,
    UDEBUG_EM_FINISHED | UDEBUG_EM_STOP | UDEBUG_EM_SYSCALL_B |
    UDEBUG_EM_SYSCALL_E | UDEBUG_EM_THREAD_B | UDEBUG_EM_THREAD_E |
    UDEBUG_EM_BREAKPOINT, UDEBUG_EM_TRAP);

thash_t threads[COUNT];
size_t copied;
size_t needed;
udebug_thread_read(session, threads, sizeof(thash_t) * COUNT, &copied, &needed);

udebug_event_t ev_type;
sysarg_t val0;
sysarg_t val1;
udebug_go(session, threads[0], &ev_type, &val0, &val1);

// Examine the event type
istate_t context;
udebug_regs_read(session, threads[0], &context);

uint8_t buffer[SIZE];
udebug_mem_read(session, buffer, 0x1000, SIZE);

// ...
udebug_end(session);
async_hangup(session);
```
Profiling

• **Run-time performance instrumentation**
  - Exact profiling
    • Triggered by specific events
    • Sampling relevant information (timestamp, CPU performance counters, stack trace, etc.)
    • E.g. GNU Profiler
      - `gcc -pg -mrecord-mcount -mno-op-mcound`
      • After instrumentation, calls `mcount()` in the given function prologues/epilogues
      • Data collected in `gmon.out`, postprocessed by `gprof`
  - Statistical profiling
    • Sampling relevant information in regular intervals
    • E.g. OProfile (system-wide profiling)
Performance Metrics

- **Resource accounting**
  - Memory (resident / virtual / shared, buffers, caches)
  - Time
    - User time, system time, idle time (precise measurements)
      - $\%\text{user} + \%\text{system} + \%\text{idle} = 100\ %$
      - Utilization = $\%\text{user} + \%\text{system}$
  - Saturation (sampled in regular intervals)
    - How much more work is there than the machine can handle without latency
      - E.g. number of non-idle CPUs + length of the scheduler ready queues
      - Usually exponential moving average: $\text{cur} = \text{prev} \times \text{decay} + n \times (1 - \text{decay})$
Performance Metrics

- **Microstate accounting**
  - Regular sampled accounting might miss activity that starts and completes between two sampling ticks
  - Logical event counters (per task/thread) might be more useful
    - Page faults (file system, executable, anonymous), interrupts, context switches, locking events, syscalls, thread latency (wait time before being scheduled), page reclamation scan rate (memory pressure indicator)
Tracing

- **Observing events**
  - Similar to debugging, but usually high-level events and no blocking
  - Similar to logging, but activated on-demand
    - System calls, kernel functions, library/user functions, logical events (context switches, sending/receiving packets, etc.), custom user space events
    - Usually asynchronous (avoiding serialization)
    - `truss(1)`, `strace(2)`, `ltrace(1)`
    - DTrace, SystemTap
DTrace Architecture

- **D** script
- **dtrace(1M)**
- **lockstat(1M)**
- **plockstat(1M)**
- **intrstat(1M)**
- **libdtrace(3LIB)**
- **dtrace(3D)**
- **intrstat(1M)**

**consumers**

**user-space provider**

**D compiler**

**communication device**

**user space**

**kernel**

**providers**

- **pid**
- **sysinfo**
- **sdt**
- **fasttrap**
- **syscall**
- **fbt**
- **usdt**

**DTrace**

**D virtual machine**
DTrace

- **Features**
  - Probe specification language (D script)
  - Probes
    - Instrumentation points
    - Ideally zero overhead when inactive, small overhead when active
  - Safety for production system
    - D virtual machine
      - No branching, no loops, no state changes (unless explicitly enabled)
    - No debug builds needed
      - Compact Type Information
  - Correlation of events, aggregate statistics
SystemTap

**Properties**

- Probe specification language (SystemTap script)
  - Preprocessed into a kernel module source → kernel module
- Probes
  - Instrumentation points
  - Ideally: zero overhead when inactive, small overhead when active
- Safety for production system not guaranteed
  - Uses ftrace and kprobes as kernel backends
- Requires debugging kernel build for extended functionality
- Correlation of events, aggregate statistics
D Language in a Nutshell

- **Probe**
  - `provider:module:function:name`
    - Shell pattern matching, empty component means “any”
    - BEGIN, END, ERROR

- **Predicate**
  - Optional integer or pointer expression serving as a guard

- **Actions**
  - List of statements delimited by semicolon, implicit default action (usually probe name printout)
  - C types and operators (conditional expression instead of branching), structures, scalar arrays, strings, associative arrays (scalar types as keys), associative arrays for statistical aggregation (`count()`, `sum()`, `avg()`, `min()`, `max()`, `quantize()`, etc.)
  - Global variables, thread-local variables (`self->`), clause-local variables (`this->`)
D Language in a Nutshell

- **Actions**
  - Access to kernel variables, state-dependent values (arguments, return value, `errno`, caller, current thread/process/working directory/CPU/user/group/timestamp, executable name, etc.)
  - `print()`, `stack()`, `ustack()`, `alloca()`, `bcopy()`, `copyin()`, `copyinstr()`, `strlen()`, `strjoin()`, `basename()`, `dirname()`, `cleanpath()`, `rand()`, `mutex_owned()`, `mutex_owner()`, `exit()`
  - Unsafe actions when explicitly enabled
    - `stop()`, `raise()`, `panic()`, `copyout()`, `copyoutstr()`, `system()`, `breakpoint()`, `chill()`
  - Speculative tracing
    - `speculation()`, `speculate()`, `commit()`
SystemTap Language in a Nutshell

- **Original motto**
  - “Painful to use, but more painful not to.”

- **Probe**
  - `provider[(arguments)].event_type[(arguments)][.name] [?]`
  - Wildcard pattern matching
  - `begin, end, error`
  - `syscall.name[.return], kernel.function("pattern")[.return], module("pattern").function("pattern")[.return], kernelstatement("pattern"), kprobe.function("pattern")[.return], kprobe.module("pattern").function("pattern") [.return], kernel.trace("pattern"), process("path").label("pattern"), timer.jiffies(n), timer.ms(n)`
SystemTap Language in a Nutshell

- **Actions**
  - List of statements delimited by whitespace, implicit default action (usually probe name printout)
  - C control structures, foreach iteration, C types and operators, member operator, structures, strings, associative arrays (scalar types as keys), aggregates (@count(), @sum(), @min(), @max(), @avg(), @hist_linear())
  - Conditional compilation, simple preprocessor macros
  - Embedded C
  - Context variables (with pretty-printers)
    - $1, $2, ..., @1, @2, ...
  - pid(), tid(), execname(), caller(), log(), printf(), sprintf(), print_backtrace(), print_ubacktrace()
  - Speculative tracing
    - speculation(), speculate(), commit(), discard()
DTrace Example

#! /usr/sbin/dtrace -s

syscall:::entry {
    @count[probefunc] = count();
    self->ts = timestamp;
    self->tag = 1;
}

syscall:::return /self->tag == 1/ {
    self->tag = 0;
    self->ts_diff = timestamp - self->ts;
    @total[probefunc] = sum(self->ts_diff);
    @average[probefunc] = avg(self->ts_diff);
}

END {
    printa("%s count=%@u sum=%@u average=%@u\n", @count, @total, @average);
}
#! /usr/bin/stap

global tag
global ts
global syscalls

probe syscall.* {  
tag[tid()] = 1  
ts[tid()] = local_clock_ns()
}

probe syscall.*.return {  
  if (!tag[tid()])  
    next  
  
tag[tid()] = 0  
ts_diff = local_clock_ns() - ts[tid()]  
syscalls[name] <<< ts_diff
}

probe end {  
  foreach (syscall in syscalls)  
    printf("%s count=%u sum=%u average=%u\n", syscall,  
            @count(syscalls[syscall]),  
            @sum(syscalls[syscall]),  
            @avg(syscalls[syscall]))
}
DTrace Example

#!/usr/sbin/dtrace -s
#pragma D option quiet

hotspot$target:::method-entry {
    self->indent++;
    self->trace = 1;
    printf("%*s -> %s.%s\n", self->indent, ",", stringof(copyin(arg1, arg2)), stringof(copyin(arg3, arg4)));
}

hotspot$target:::method-return /self->trace == 1/ {
    printf("%*s <- %s.%s\n", self->indent, ",", stringof(copyin(arg1, arg2)), stringof(copyin(arg3, arg4)));
    self->indent--;
    self->trace = (self->indent == 0) ? 0 : self->trace;
}

pid$target:libc::entry /self->trace == 1/ {
    self->indent++;
    printf("%*s => %s\n", self->indent, ",", probefunc);
}

pid$target:libc::return /self->trace == 1/ {
    printf("%*s <= %s\n", self->indent, ",", probefunc);
    self->indent--;
}

syscall:::entry /self->trace == 1/ {
    self->indent++;
    printf("%*s :> %s\n", self->indent, ",", probefunc);
}

syscall:::return /self->trace == 1/ {
    printf("%*s <: %s\n", self->indent, ",", probefunc);
    self->indent--;
}
Comparison

- **DTrace**
  - Available in Solaris, Illumos, macOS, FreeBSD, NetBSD
  - Installable extension for Linux, Windows

- **SystemTap**
  - Linux specific
  - Weaker backwards compatibility and maintenance across kernel versions

- **bpftrace**
  - Linux specific, architecturally more similar to DTrace
  - Some practical limitations (e.g. no user stack traces without frame pointers)
### DTrace Code Instrumentation

<table>
<thead>
<tr>
<th>uninstrumented</th>
<th>instrumented</th>
</tr>
</thead>
</table>
| `squeue_enter_chain+0x1af:` | `xorl %eax,%eax`  
`squeue_enter_chain+0x1b1:` | `nop`  
`squeue_enter_chain+0x1b2:` | `nop`  
`squeue_enter_chain+0x1b3:` | `nop`  
`squeue_enter_chain+0x1b4:` | `nop`  
`squeue_enter_chain+0x1b5:` | `movb %bl,%bh`  
`squeue_enter_chain+0x1b6:` | replaced by the call instruction |
|                         | `xor %eax,%eax`  
`ufs_mount:` | `nop`  
`ufs_mount+1:` | `nop`  
`ufs_mount+4:` | `nop`  
`ufs_mount+0xb:` | `lock nop`  
|                         | `nop`  
`ufs_mount+0x3f3:` | `movb %bl,%bh`  
`ufs_mount+0x3f4:` | `int $0x3`  
`ufs_mount+0x3f7:` | `movq %rsp,%rbp`  
`ufs_mount+0x3f8:` | `subq $0x88,%rsp`  
|                         | `pushq %rbp`  
                         | `subq $0x88,%rsp`  
                         | `pushq %rbx`  
                         | `popq %rbx`  
                         | `movq %rbp,%rsp`  
                         | `popq %rbp`  
                         | `ret`  
                         | `int $0x3`  

ftrace Code Instrumentation

- **Using gcc -pg to call __fentry__() in every prologue**
  - Patched out at kernel load time

<table>
<thead>
<tr>
<th></th>
<th>Uninstrumented</th>
<th>Instrumented</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit_mmap:</td>
<td>nop</td>
<td>int $0x3</td>
</tr>
<tr>
<td>exit_mmap+1:</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>exit_mmap+2:</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>exit_mmap+3:</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>exit_mmap+4:</td>
<td>nop</td>
<td>nop</td>
</tr>
</tbody>
</table>

1: replaced by the address
2: replaced by the call opcode
Code Instrumentation

- **Static instrumentation of executables**
  - Non-intrusive
    - Avoiding shifting instruction locations (complicated unless the code is position-independent)
    - Replacing a byte of an instruction by a trap instruction
      - At run-time, replacing the trap with a call
      - The call emulates the original instruction(s) that are replaced and jumps back to the next instruction
  - Intrusive
    - Techniques similar to binary translation
    - Internal representation of code basic blocks
      - Instrumentation is equal to inserting a new back block and updating jump/call locations
      - Tricky with self-modifying code, dynamic dispatch, etc.
  - Valgrind, DynInst, Pin, Vulcan, BIRD, PEBIL
Post-Mortem Analysis

- **Analyzing a root cause of a crash**
  - Core dump
    - Snapshot of a single process
    - On-disk format similar to an executable format
      - Added state/register context and other metadata
      - Can be opened in an interactive debugger
  - Crash dump
    - Snapshot of an entire system
      - Sometimes without user pages and other sensitive data
      - Created by a failing system, rescue system (kexec), firmware or out-of-band management
      - Non-maskable interrupt
      - Used by special analysis tools
Core/ Crash Dump Analysis

- **Identifying the immediate cause**
  - Examining the crash IP location, register context, stack trace, log buffer, instrumentation values (if available)

- **Identifying the root cause**
  - Art, science and craft
    - Values of registers (esp. scratch) and arguments can be lost or misleading
    - Control flow only partially obvious from stack traces (at best, if frame pointers are used)
    - The more the code is optimized (leaf calls, tail calls, inlining) the worse it is to understand (usually)
  - Heuristic tools to analyze typical crashes
    - ::findlocks in mdb for Solaris
    - Crash tool for Linux
  - Gradually reconstructing the events prior to the crash
    - Formulating hypotheses while distrusting the information encountered
    - Analyzing data structures, threads, locks, etc.
    - Looking for interesting literals (0xdeadbeef, 0xbaddcafe, 0xfeedface)
References


Thank you!

Questions?