Tracing Frameworks

Jack Henschel

March 2017

Introduction

Tracing Frameworks provide a direct interface to inspect, test, debug and measure running applications (so called "online code"). This can be valuable while developing a program (e.g. for performance analysis or error checking), but also when troubleshooting issues after deployment in the field. Some frameworks don't require modification of application source code at all, others depend on entry points ("markers") or loading additional libraries.

In this paper I will demonstrate the usage of SystemTap and evaluate two other tracing frameworks (Frida and LTTng) at the end.

1 SystemTap

SystemTap is developed by Red Hat Inc. and had its initial release in 2005. The tool has its own scripting language with a similar syntax to C, which consists of global variables, functions and probe point definitions. These "stap scripts" are then parsed, resolved (for symbols, types and probes) and translated into C code by SystemTap. From there on SystemTap uses one of its (currently) two supported backends: the Linux kernel module backend and the DynInst library backend.

The Linux kernel backend works by compiling the C code into a loadable kernel object module (extension *.ko*), which can then be loaded and executed by the Linux kernel. This backend has the power to probe any code running in kernel- or user-space mode. However, loading a kernel module requires elevated privileges.

Alternatively, the DynInstAPI backend uses a dynamic program analysis approach for binary instrumentation, analysis and modification. This backend can be used without root privileges, however can only probe code executed by the same user. This backend offers even lower overhead than the Linux kernel module backend, was however to unstable in testing to be evaluated further (application crashes through memory corruption).

Here is an example of a basic stap script:

```
#!/usr/bin/stap

probe kernel.module("vfs").function("read") {
  printf("read from the virtual filesystem\n")
  exit()
}
```

SystemTap's scripting language makes it both very flexible and easy to use.

However, the Linux kernel backend of SystemTap does also have a downside: it requires various kernel headers and debug symbol objects to be installed on the system, so SystemTap knows where to place its probes in the code segment in the memory. Red Hat and Debian based distributions provide these as debug packages (package suffix *dbg* or *dbgsym*) which can easily be installed via the distribution's package manager. In custom built environment it can be troublesome to obtain these files.

To illustrate the possibilities of SystemTap, I will use the following C program.

simple.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <limits.h>

static long double factors[2];

static int isSquareNumber(long long n) {
    long long i = 1;
    while(n > 0) {
        n -= i;
        i += 2;
    }
    return n == 0;
}

static void factor(unsigned long long n) {
    long double r, x, y;

    x = ceill( sqrtl(n) );
    r = x*x - n;
    while ( ! isSquareNumber(r) )
        r += 2*(x++) + 1;
    y = sqrtl(r);
    factors[0] = x + y;
    factors[1] = x - y;
}

int main(int argc, char **argv) {
    char *t;
```

```
    unsigned long long n;

    if ( argc != 2 ) {
        printf("Usage: %s <UNSIGNED INTEGER>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    errno = 0;
    n = strtoull(argv[1], &t, 10);
    if ( ! t || (n == ULLONG_MAX && errno) ) {
        perror("strtoull");
        exit(EXIT_FAILURE);
    }

    factor(n);

    printf("%lld = %Lf * %Lf\n", n, factors[0], factors[1]);

    exit(EXIT_SUCCESS);
}
```

This C program will spend most of its time in the function `factor` and jump to the function `isSquareNumber` very often. These are two interesting metrics one can measure with tracing frameworks.

Using a SystemTap script ("stap script") we can measure these metrics. First off, some global variables to store values in:

perf.stp (Part 1)

```
global process_exec_time
global func_factor_time
global func_factor_cycles
global func_isSquareNum_calls
```

Next, the actual probe points (these are the places in the code where SystemTap will be triggered via a software breakpoint):

perf.stp (Part 2)

```
/* save starting time of given process */
probe process.begin {
    process_exec_time = gettimeofday_ns()
}

/* calculate time difference from start time and save it */
probe process.end {
    process_exec_time = gettimeofday_ns() - process_exec_time
}

/* increment counter for each call of isSquareNumber */
probe process.function("isSquareNumber").call {
    func_isSquareNum_calls <<< 1
}
```

```
/* save starting time and cycles of function factor */
probe process.function("factor") {
    func_factor_time = gettimeofday_ns()
    func_factor_cycles = get_cycles()
}

/* calculate time difference from start time and save it */
probe process.function("factor").return {
    func_factor_cycles = get_cycles() - func_factor_cycles;
    func_factor_time = gettimeofday_ns() - func_factor_time
}
```

As can be easily seen, we are measuring the total execution time of the process (with `process.begin` and `process.end`), the time and CPU cycles spent in the function `factor` (with `process.function("factor")` and `process.function("factor").return`), as well as the number of times the function `isSquareNumber` was called (with `process.function("isSquareNumber").call`) and a special operator called *statistical aggregate*.

The *statistical aggregate* operator in SystemTap is made for storing data very fast. Unlike variables and arrays, it does not lock the variable when writing to it, which makes the operation very "cheap". However, when reading data from the statistical aggregate, one must not expect the values to have any kind of order (i.e. chronological order is not guaranteed).

At the end of the probe (or when the process finishes), we are collecting the results of the measurements and display them to the user:

perf.stp (Part 3)

```
/* print statistics when stap exits */
probe end {
    printf("Execution time of Process %d: %d ns\n",
           target(), process_exec_time)
    printf("Execution time of Function factor: %d ns\n",
           func_factor_time)
    printf("Total cycles in Function factor: %d\n",
           func_factor_cycles)
    printf("Total calls of Function isSquareNumber: %d\n",
           @count(func_isSquareNum_calls))
    delete process_exec_time
    delete func_isSquareNum_calls
    delete func_factor_time
    delete func_factor_cycles
    exit()
}
```

It outputs the process id (PID) of the traced program along with the total runtime of the process, the time and CPU cycles spent in the function `factor` and how often the function `isSquareNumber` was called. Afterwards, the values for the items are deleted and SystemTap exits.

But how does SystemTap know *which* process it should attach to? In the SystemTap script shown above I simply used the probe point `process`,

but one can also specify an explicit process path (such as `process("./myapp")` or `process("/lib/liberty.so")`) or process id (`process(123)`).

If no process is defined in the script, one has to provide this information to SystemTap via command line. This way, one can write generic scripts which can attach to any arbitrary process the users chooses.

After compiling the test application to the binary `simple`, we can run the SystemTap script `perf.stp`. This may look as follows:

```
$ gcc -std=gnu99 -lm -o simple simple.c
$ stap -c './simple 129129637' perf.stp
129129637 = 83471.000000 * 1547.000000
Execution time of Process 4170: 2598279897 ns
Execution time of Function factor: 2577533391 ns
Total cycles in function factor: 6222798641
Total calls of Function isSquareNumber: 31146
```

The first line of output comes from the program itself, the application calculated 83,471 and 1547 as the two factors of 129,129,637. The following lines were output by SystemTap: the process ran with PID 4170 and had a total execution time of about 2.6 seconds (2,598,279,897 ns). As anticipated, most of this time was spent in the function `factor`, namely 2.58 seconds (2,577,533,391 ns) or over six billion CPU cycles (6,222,798,641). The `isSquareNumber` function was called over thirty thousand times (31,146).

Of course, we already knew these results beforehand. But it just goes to show that the methodology actually works.

Variables SystemTap can also read arbitrary variables from any place in the program, including return values and function parameters.

Assuming we have a function named `example` with a function parameter `char *message` (essentially containing a string), we can access this variable by simply prefixing it with a dollar sign:

```
/* void* example(char *message) */
probe process.function("example") {
  if($message != 0) {
    printf("message: %x\n", $message)
  }
}
```

Of course, this is still pretty useless, because we want the message itself and not the address of the pointer to the message (although, there may be various cases where this information could be valuable, too). To dereference a pointer in a SystemTap script, we first need to know if we are dealing with a variable in kernel- or user-space, then we can choose the appropriate

function (see Section 3.3.2 and 4.2 of the SystemTap Beginners Guide for more information).

```
/* void* example(char *message) */
probe process.function("example") {
  if($message != 0) {
    m = user_string(message)
    printf("message: %s", m)
  }
}
```

For kernel addresses, the corresponding functions are called `kernel_*` instead of `user_*` and the same functions are available.

To simply print all variables, function parameters, local variables or the return value of a function, SystemTap has special operators: `$$vars`, `$$locals`, `$$parms` and `$$return`. These expand to a string equivalent of `sprintf("var1=%x var2=%x ... parm1=%x ... local1=%x ...", var1, var2, ..., parm1, ..., local1, ... ")`.

```
probe kernel.function("vfs.read") {
  printf("%s\n", $$vars)
}
```

For an in-depth explanation of pretty printing target variables, please consult SystemTap Beginners Guide Section 3.3.2.1.

Functions The more complex one's stap scripts become, the more useful are SystemTap functions. These are declared by the prefix `function` and optional function parameters. Function parameters may be typed by suffixing the parameter name with `:type`.

```
/* swap bytes of a short (16 bit) */
function swap_short (addr:long) {
  first = user_char(addr + 0)
  second = user_char(addr + 1)
  number = first << 8 | second
  return number
}
```

Overhead Live probing any application comes with executing additional code, i.e. overhead. It is very important to gain an understanding of the performance hit that comes with these measurements. In order to do so, I wrote a C application which measures its own runtime and does not do any other processing.

time.c

```c
#include <stdio.h>
#include <time.h>

#define SAMPLES 10000

void func_call(struct timespec *tp) {
    asm("");
    clock_gettime(CLOCK_MONOTONIC, tp);
    asm("");
}

void func_return(struct timespec *tp) {
    asm("");
    clock_gettime(CLOCK_MONOTONIC, tp);
    asm("");
}

void func_total(struct timespec *tp) {
    asm("");
    clock_gettime(CLOCK_MONOTONIC, tp);
    asm("");
}

int main() {
    /* measure overhead of "clock_gettime" call */
    struct timespec tp;
    long nsecs[SAMPLES];
    long avg;

    for(int i = 0; i < SAMPLES; i++) {
        asm("");
        clock_gettime(CLOCK_MONOTONIC, &tp);
        asm("");
        nsecs[i] = tp.tv_nsec;
    }

    for(int i = SAMPLES-1; i > 1; i--) {
        avg += nsecs[i] - nsecs[i-1];
    }
    avg /= SAMPLES-1;
    printf("Average Clock Call Overhead: %ld ns\n", avg);

    /* measure overhead of probes */
    struct timespec before, in, after;
    long avg_delta_total = 0, avg_delta_call = 0,
        avg_delta_return = 0;
    for(int i = 0; i < SAMPLES; i++) {
        /* measure total function runtime */
        clock_gettime(CLOCK_MONOTONIC, &before);
        func_total(&in);
        clock_gettime(CLOCK_MONOTONIC, &after);
        avg_delta_total += after.tv_nsec - before.tv_nsec;
```

```
        /* measure function call time */
        clock_gettime(CLOCK_MONOTONIC, &before);
        func_call(&in);
        avg_delta_call += in.tv_nsec - before.tv_nsec;

        /* measure function return time */
        func_return(&in);
        clock_gettime(CLOCK_MONOTONIC, &after);
        avg_delta_return += after.tv_nsec - in.tv_nsec;
    }
    avg_delta_total /= SAMPLES;
    avg_delta_call /= SAMPLES;
    avg_delta_return /= SAMPLES;

    printf("Average Total: %ld ns\nAverage Jump In (Call): %ld
        ns\nAverage Jump Out (Return): %ld ns\n",
            avg_delta_total, avg_delta_call, avg_delta_return);

    return 0;
}
```

This program first measures the average time it takes for a `clock_gettime` call. Naturally, the results will vary from machine to machine, nevertheless we should be able to see the relative performance impact SystemTap has on the application. The various empty inline assembler statements (`asm ("")`) prevent the compiler from doing any unwanted optimization (such as function inlining).

These are the results on my machine:

```
$ gcc --std=gnu99 -o time time.c
$ ./time.c
Average Clock Call Overhead: 113 ns
Average Total: 158 ns
Average Jump In (Call): 78 ns
Average Jump Out (Return): 79 ns
```

Next up, we place probes at the start and end of the function calls and observe the differences.

probes.stp

```
probe process.function("func_total") {}
probe process.function("func_call").call {}
probe process.function("func_return").return {}
```

This stap script places probes at the beginning of the `func_total` function, at the beginning of the `func_call` function (the `.call` suffix is optional and can be omitted) and at the end of the `func_return` function (`.return`).

Because these probes are empty SystemTap will elide them by default. To avoid this, we need to use the *unoptimized mode* via the `-u` command-line

switch.

```
$ stap -u -c ./time.c probes.stp
Average Clock Call Overhead: 83 ns
Average Total: 4200 ns
Average Jump In (Call): 4143 ns
Average Jump Out (Return): 800 ns
```

These measurements show us that there is a significant hit for placing probes inside the application code. Jumping into the function took about 50 times longer and returning from it 10 times longer. Also keep in mind that return probes are essentially placed by putting a probe at the beginning of the function and then putting another one at the end. This means the overhead for a return probe is the cost of "Jump In" and "Jump Out" combined.

Nevertheless, these results are decent considering the power and flexibility SystemTap offers for inspecting code on-the-fly, as we've seen before.

2 Frida

Frida is a very young tracing framework for Windows, macOS, Linux, iOS, Android and QNX. Being written in JavaScript, Python and C it is rather high-level, which on one hand makes it very powerful and easy to use, but also too slow for real-time performance analysis. Another disadvantage are its dependencies (due to being written partly in JavaScript and Python).

3 LTTng

LTTng allows tracing of applications written in C, C++, Java and Python on a Linux platform, as well as the Linux kernel itself (due to being written in C).

It is designed from the ground up to provide low overhead tracing on production systems and offers the possibility of recording traced events. Additionally, these can be exported to other tools like LTTng analysis or Babeltrace and even graphically viewed with Trace Compass, which is very handy and exactly what one wants for performance analysis.

One downside of LTTng is it requires modification of the source code, because auxiliary markers for entry points and the LTTng shared library need to be loaded. While some applications (like the Linux kernel) already come with these markers and can be used right away with LTTng, this is not the case for our target application. However the biggest obstacle is the additional library for LTTng, which would need to be present on every system.