Functracer Internals

COLLABORATORS			
	TITLE : Functracer Internals		*REFERENCE* :
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Anderson Lizardo and Bruna Moreira	2008-12-04	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.1	2008-12-04		AL

Contents

List of Figures

Abstract

This document describes Functracer internals, and is intended for those interested in getting a better understanding on how it works “behind the scenes”. It also serves as a guide for reading the code and making modifications to it.

This document describes the functionality of functracer versions greater than or equal to 0.13.

1 Introduction

Functracer is a specialized, tracing-based debugging tool. It works by attaching to a running process (using the **-p** option), or by executing a new program passed to it as argument.

It is extensible by the means of plugins, that allow performing actions, such as collecting function call stack traces (also known as “backtraces”) and reporting events like process creation or destruction and calls to library functions.

One of these plugins is a memory usage tracer, that tracks memory related library function calls (such as `malloc()`, `calloc()`, `realloc()` etc.) and generates backtraces of these calls.

This document will focus on describing functracer behavior only on ARM and x86 architectures, given that they are the only architectures supported at the moment.

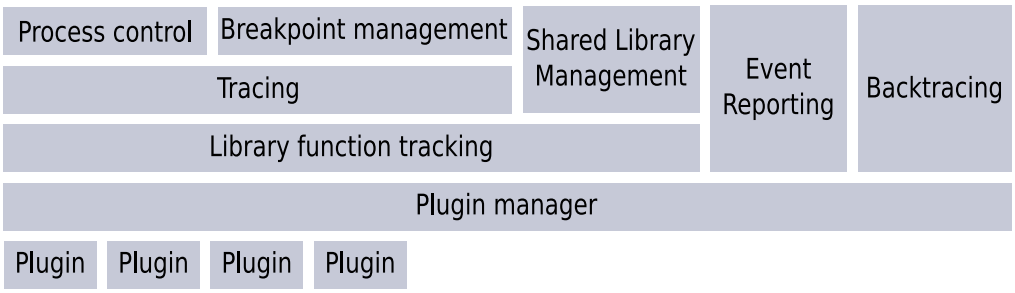

Figure 1: Functracer components.

2 Functracer components

Functracer is composed of a number of components, namely:

- Process control
- Breakpoint management
- Tracing
- Shared library management
- Library function tracking
- Event reporting
- Backtracing
- Plugin manager

Each of these components will be described in the following sections.

2.1 Process control

This component uses the ptrace API to attach to processes and to control their execution. On each low-level event (e.g. a signal received by a process, a process is created/destroyed, a `fork()`, `clone()` or `exec*()` call is made) the process is interrupted, allowing functracer to inspect the current process state. For a full list of low-level events handled by functracer, see the `struct event` definition.

Each traced process is represented in functracer by a struct process. The underlying ptrace API works only with process IDs (PIDs), therefore functracer is responsible for matching already seen processes with existing struct process instances.

2.2 Breakpoint management

Functracer uses regular software breakpoints (referred as just “breakpoints” on this document) to interrupt process execution at specific addresses and inspect its state. Breakpoints can be inserted or removed from a process, and the original instruction located at the address is saved or restored where necessary.

On ARM, a breakpoint is an undefined instruction with a specific encoding that is guaranteed to never be used in future extensions to the instruction set. On x86, a breakpoint is the "int3" instruction, reserved for software breakpoints.

Functracer manages breakpoints using `struct breakpoint` instances, that are stored in a dictionary indexed by the memory address where it is located. This structure holds, amongst other things, the breakpoint instruction itself (which, in case of ARM, can be one of two values, depending on whether the CPU is in ARM or Thumb mode at the moment the instruction will execute) and the original instruction located at the address where the breakpoint will be inserted.

Breakpoints are shared amongst threads, because they share the same memory address space. This requires additional care to avoid race conditions where, for instance, a breakpoint is removed in a thread, while another thread is about to execute the instruction at that address, thus missing the breakpoint. To avoid this issue, the concept of "single-step out of line" (or SSOL for short), borrowed from the Kprobes implementation in the Linux kernel, was implemented in functracer. The SSOL mechanism will be described in the following section.

2.2.1 Single-Step Out of Line (SSOL)

The SSOL concept adopted by functracer was based on a similar idea discussed in the SystemTap mailing list [1]. Whereas in the email the technique was designed to be implemented in kernel code, the basic concept can be implemented in user space.

The SSOL approach differs from traditional breakpoint management (such as that used by ltrace and by earlier functracer versions) in that the breakpoint does not need to be replaced with the original instruction to be executed. To achieve that, the original instructions are stored in a area of the process address space reserved by performing an artificial `mmap()`, called “SSOL area”, and whenever the breakpoint is hit, the process instruction pointer is changed to the location of the original instruction in the SSOL area, the instruction is single-stepped and then the instruction pointer is changed to the instruction following the breakpoint.

One of the side-effects of this approach is that, for breakpoints that are never disabled, it is not required to temporarily disable them, which on traditional breakpoint scheme would allow for a concurrent thread to pass over the breakpoint without hitting it. SSOL-based breakpoints are never disabled (except when detaching from a process), thus all threads will catch them.

This approach works well for function entry breakpoints, but for return breakpoints it would require many entries (one per return address) and managing them would be complex. See “Library function tracking” for a description on how return breakpoints are handled in functracer.

2.3 Shared library management

Most functracer plugins track functions exported by shared libraries. These can be loaded during application startup (e.g. when the library is dynamically linked to the application), or on demand using a `dlopen()` call. Both methods are properly handled by functracer.

[1] http://sourceware.org/ml/systemtap/2007-q1/msg00571.html

The list of currently loaded shared libraries is read from `/proc/PID/maps`, and functions that manipulate this file are in `src/maps.c`. The maps entries that do not contain executable code (indicated by its permissions flags) is filtered out, and the remaining entries are inserted into a `struct solib_list` instance. This is done once when functracer attaches to a process (to collect early linked libraries such as the dynamic linker itself), and the list is updated for each new library loaded or unloaded.

The dynamic linker exposes a symbol called "_dl_debug_state" used specifically for instrumenting the loading and unloading of shared libraries. It works by inserting a breakpoint at the address pointed by the symbol, so on every library load or unload, the breakpoint is hit, and `/proc/PID/maps` can be re-examined.

2.4 Library function tracking

With the support of the components described until now, it is possible to track specific library functions by inserting breakpoints at their entry point (to examine function arguments) and return address (to examine the return value, if any).

The entry breakpoints for all functions to be tracked are inserted as soon as the library that exports it is loaded. The return breakpoints, on the other hand, can only be inserted after the entry breakpoint is hit, because the actual return address will be known only when the function is called.

Given that a function can be called from multiple locations, having a return breakpoint for each function call can quickly increase the number of breakpoints enabled concurrently. To keep the number of breakpoints manageable, functracer uses only one breakpoint for all function returns, and instead of inserting breakpoints at the return addresses, it modifies the return address of the function to point to that shared breakpoint. That return address is then "fixed" when the breapoint is hit, and program execution is redirected to the original address.

Function calls are tracked with the help of a stack structure, called `struct callstack`. This structure holds state information saved on function entry, such as registers, that are then used at function exit to get function arguments and restore the original return address.

2.5 Event reporting

Events are reported in a custom format, that can be easily parsed by post-processing tools such as functracer-postproc. The trace format is described in the README file that comes with functracer source.

The event reporting functions are usually called by plugins, and each process has its own trace file (threads share a single trace file).

2.6 Backtracing

Backtraces are very useful to locate a specific event in the sources of the target application, and to analyze the context where the event occurred. It consists of a stack of function names, linked by a "called by" relation, with the base (i.e. the last function in the backtrace) usually being the main() function, the exception being when the backtrace is generated from a signal handler.

Backtraces are generated in functracer with the help of the libunwind library. Libunwind supports "remote" backtracing (e.g. when the process requesting the backtrace is different from the one for which the backtrace is generated) natively using the ptrace API.

2.7 Plugin manager

Functracer has support for plugins, that allow collecting useful program state when specific events occur. Currently, functracer is able to handle the following events:

- A function is called
- A function returns to its caller
- A process is created
- A process executes a program (using only of the `exec*()` functions)

- A process exits (normally or killed by a signal)
- A process receives a signal
- A process is “interrupted” because functracer is exiting
- A system call is made (currently disabled due to performance impact)
- A system call returns (currently disabled due to performance impact)
- A new symbol is exported by a library function (useful for selecting functions to be tracked)

On the occurrence of the above events, the following data can be collected:

- Function arguments
- Function return value
- Process ID (PID)
- Program executed by a `exec*()` call
- Exit code
- Signal received

A plugin consists of a shared object that exports a single function called `init()`. This function returns a instance of `struct plg_api` containing pointers to functions to be called when one of the events above occurs.