Contents

1 Introduction

Self-propelled instrumentation is a binary instrumentation technique that dynamically injects a fragment of code into an application process on demand. The instrumentation is inserted ahead of the control flow within the process and is propagated into other processes, following communication events, crossing host boundaries, and collecting a distributed function-level trace of the execution.

Self-propelled instrumentation contains two major components, *Agent* and *Injector*. *Agent* is a shared library that automatically inserts and propagates a piece of payload code at function call events in a running process, where the payload code contains user-defined logic, such as generating trace data for later inspection. The instrumentation would propel itself within the process by following control flow and across thread boundaries, process boundaries, or even host boundaries by following communication flow. *Injector* is a process that causes an application process to load the Agent shared library, where the Injector should have at least the same privilege as the application process. Self-propelled instrumentation does binary instrumentation within the application process's address space, avoiding use of the debugging interfaces (e.g., Linux ptrace and Windows debug interface) and costly interprocess communications. Therefore, self-propelled instrumentation does not add significant overhead to a process during runtime.

Self-propelled instrumentation can be used in many applications that require low overhead instrumentation and full automation of instrumentation propagation following control flow. For example, we have used self-propelled instrumentation for problem diagnosis in distributed systems [?] and for automated diagram construction for complex software systems in security analysis [?].

2 Abstraction

Self-propelled instrumentation has two major components, *Agent* that is a shared library injected into an application process's address space, and *Injector* that injects *Agent*. The following subsections describe the lower level components in Agent and Injector in details.

2.1 Agent

- Agent. It manages the configuration and does instrumentation. An Agent instance is created in the init function of the *Agent* shared library.
- Event. It specifies what kind of initial instrumentation should be done once the *Agent* shared library is loaded. Currently, there are three types of Event: 1) instrumenting all

callees in *main* function right away; 2) instrumenting all callees of specified functions right away; 3) instrumenting specified function calls right away.

- Payload function. It contains user-specified code. Frome user's perspective, a payload function will be invoke before or after each function call in the process.

- Point. It represents an instrumentation point at current function call and is used in Payload function.

- Control Flow Graph (CFG) structures. CFG structures include Object, Function, Block, and Edge. An Object represents a binary file (i.e., an executable or a shared library), and contains a set of functions. A Function contains a set of Blocks. A Block is a basic block. An Edge connects two Blocks. Users can get related CFG structures of current function call from Point.

- AddressSpace. It represents the address space of the process. It contains a set of Objects in the process. Also, it implements some memory management primitives used by the instrumentation engine.

- Parser. It represents a binary code parser that parses binary code into structural CFG structures, i.e., Object, Function, Block, and Edge.

- Propeller. It manages intra-process instrumentation propagation, where it finds function call Points inside current function and uses Instrumenter to insert Snippets at these points.

- Snippet. It represents a patch area that contains function calls to the Payload function and the relocated function call or the relocated call block.

- Instrumenter. It is the instrumentation engine that uses a set of Instrumentation Workers to insert Snippets to function call points.

- Instrumentation Worker. It represents a mechanism of installing instrumentation. Currently, four types of Instrumentation Workers are implemented: 1) relocating original function call instruction; 2) relocating original call block; 3) relocating nearby large springboard block; 4) using trap instruction.

- IpcMgr. It manages inter-process instrumentation propagation by creating Channels and using IPC Workers.

- Channel. It represents a unidirectional communication channel, containing local process name and remote process name.

- IPC Worker. It implements inter-process instrumentation propagation for a particular IPC mechanism (e.g., TCP, UDP, pipe).

2.2 Injector

Injector is provided as a command. There are two types of injections. One is to inject the *Agent* shared library at the very beginning of a process. The other is to inject the *Agent* in the middle of a running process.

The first type of Injector relies on dynamic linker (i.e., setting the environment variable LD_PRELOAD to the path of an *Agent* shared library). The second type uses ProcControlAPI to force an application process to invoke functions in the dlopen family.

3 How it works

This section describes how self-propelled instrumentation works in details. Each subsection is a major step in the workflow. Thw workflow is visualized in Figure **??**.

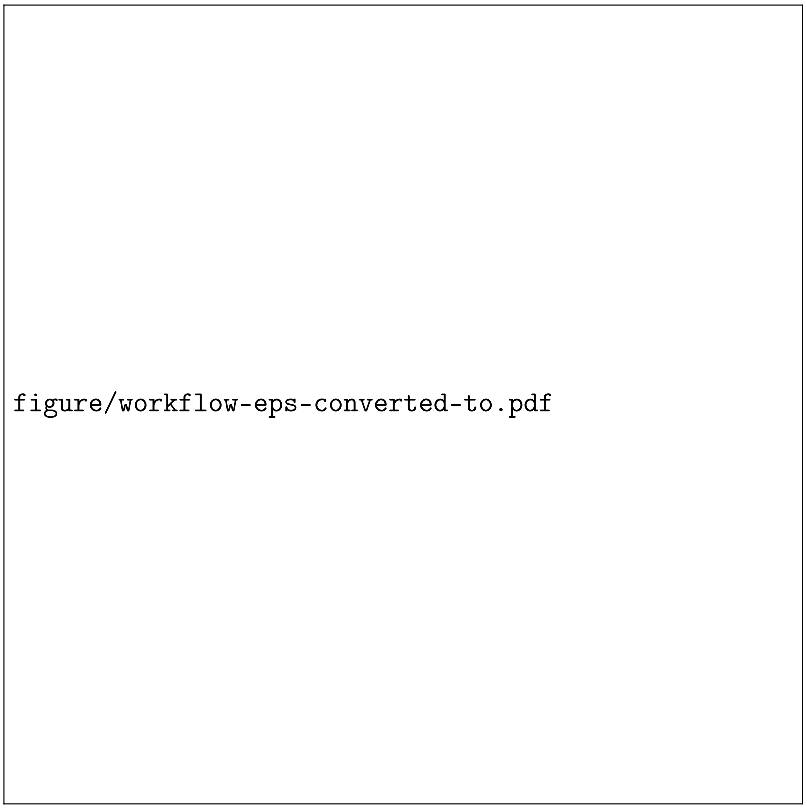

Figure 1: Self-propelled Instrumentation Workflow

3.1 Building Agent

Users build their own *Agent* shared library using self-propelled instrumentation's API.

1. Coding. Users need to write two pieces of code: 1) payload function; 2) configuration code that registers payload function and does some customization and configuration. The configuration code must be executed right away when the *Agent* shared library is loaded into the application process, so the configuration code should be in the init function of the *Agent* shared library, i.e., the function with gcc directive __attribute__((constructor)).

2. Building. Users build the code into an *Agent* shared library linking with *libagent.so* provided by the self-propelled instrumentation infrastructure.

3.2 Injection

Users run *Injector* in command line. They specify in command line arguments the path of an *Agent* shared library and the application process to inject to.

One trick to check whether the *Agent* shared library is injected successfully is to look at memory maps file of the application process, i.e., /proc/PID/maps.

3.3 Configuration

The configuration code is executed right away when *Agent* shared library is load into the application process. It tells self-propelled instrumentation what are payload functions provided by users, how would initial instrumentation be done, whether or not to enable interprocess instrumentation propagation ...

3.4 Initial Instrumentation

3.5 Intra-process Propagation

3.6 Inter-process Propagation

4 Examples

4.1 Writing Payload

Listing 1: Writing Payload

```
void payload(SpPoint* pt) {
  SpFunction* func = sp::Callee(pt);
  if (func == NULL) return;
  sp::Propel(pt);
}
```

4.2 Configuration

Listing 2: Configuration

```
TODO
```

4.3 Using Injector

4.4 Extending SPI

4.4.1 Parser

4.4.2 Event

4.4.3 AddrSpace

4.4.4 CFG

4.4.5 Point

4.4.6 Instrumentation Workers

4.4.7 IPC Workers

5 Class Reference

5.1 Public Interface

5.1.1 Class Agent

5.1.2 Class SpPoint

5.1.3 Class SpObject

5.1.4 Class SpFunction

5.1.5 Class SpBlock

5.1.6 Class SpEdge

5.1.7 Utility Functions

5.2 Private Interface

5.2.1 Class SpAddrSpace

5.2.2 Class IpcMgr

A Installation

B Testing

C Directory Organization

- x86-unknown-linux-2.4/
- i386-unknown-linux-2.4/