Paradyn Parallel Performance Tools

Self-propelled Instrumenta

Developer's Manual

1.0b Release
September 2012

Computer Science Department
University of Wisconsin–Madison
Madison, WI 53711

Computer Science Department
University of Maryland
College Park, MD 20742

Email bugs@dyninst.org
Web www.dyinst.org

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 **??**.

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 inter-process 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

```
1 void payload(SpPoint* pt) {
2   SpFunction* func = sp::Callee(pt);
3   if (func == NULL) return;
4   sp::Propel(pt);
5 }
```

4.2 Configuration

Listing 2: Configuration

```
1 TODO
```

A Installation

This appendix describes how to build SecSTAR from source code, which can be downloaded from `http://www.paradyn.org` or `http://www.dyninst.org`.

BUILDING ON UNIX
Before starting to build SecSTAR, you have to make sure that you have already installed and built Dyninst. (refer http://www.dyninst.org/sites/default/files/manuals/dyninst/dyninstProgGuide.pdf - Appendix D for building Dyninst)

Building SecSTAR on UNIX platforms is a very simple three step process that involves: unpacking the SecSTAR source, configuring paths in config.mk, and running the build.

1. Unpacking the SecSTAR source:
SecSTARâĂŹs source code is packaged in a —(tar.gz) format. If your SecSTAR source tarball is called *src_SecSTAR.tar.gz*, then you could extract it with the commands *gunzip src_SecSTAR.tar.gz ; tar âĂŞxvf src_SecSTAR.tar*. This will create a list of directories and files.

2. Configuring paths in config.mk
After unpacking,the next thing is to set DYNINST_ROOT, PLATFORM, and LD_LIBRARY_PATH environment variables in.
*D*YNINST_ROOT should be set to path of the directory that contains subdirectories like dyninstAPI, parse API etc., i.e. within dyninst directory SP_DIR should be set to the the path of the current working directory (where self propelled instrumentation is installed).

$DYNLINK should be set true for building agent as a small shared library that relies on other shared libraries. Otherwise, a single huge shared library that static-linked all libraries(doubt)

$PLATFORM should be set to one of the following values depending upon what operating system you are running on:

- i386-unknown-linux 2.4: Linux 2.4/2.6 on an Intel x86 processor
- x86_64-unknown-linux2.4: Linux 2.4/2.6 on an AMD-64/Intel x86-64 processor

Before building, you should also check whether $LD_LIBRARY_PATH variable is set. If you are using bash shell, then open /.bashrc file and check if $LD_LIBRARY_PATH is already present. If not, then LD_LIBRARY_PATH variable should be set in a way that it includes $DYNINST_LIB PATH directories. If you are using C shell, then do the above

mentioned tasks in /.cshrc file.

3. Building SecSTAR:
Once config.mk is set, you are ready to build SecSTAR. Move to $PLATFORM directory and execute the command *make*. This will build ——(DyninstâĂŹs mutator library, the Dyninst runtime library, and DyninstâĂŹs test suite). —- (Successfully built binaries will be stored in a directory named after your platform at the same level as the dyninst directory)

B Testing

C Directory Organization

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