Self-Propelled Instrumentation developing notes

libagent.so:

Self-Propelled Instrumentation generates this shared library. It contains the core functionalities.

In linux, when shared library is loaded, its init function gets called. We take advantage of this behavior and kick start the instrumentation in the init function of the agent library. We let users to write their own AgentGo function and declare it as the init function of the library. At the end of the init function, the go function is expected to get called and kick off the initial setup and instrumentation.

The fini function could also be declared, but it is less reliable since the process can end unexpectedly.

Agent:(agent.cc)

This class is mainly responsible for initializing the instrumentation.

In the SpAgent::Go() function:
1. Survey environment variables and do the configurations.
2. Initialize SpParser, parse and create CodeObject for each code object in the program.
3. Initialize event.
4. Initialize SpContext, it will use the stackwalker to find functions on the stack in the case of injection.
5. Initialize AddrSpace.
6. Check if user specifies the payload functions. If not, use default payload.
7. Instrument some special functions at first. (Ex: exit in libc)
8. Register event.

PointCallHandle: (payload.cc)

This is a newly introduced concept. This represents a call that happens at a particular time at a particular Point. It is different from the Point's abstraction. A Point is a static location in the binary code, while the PointCallHandle represents a call event that happens at the point. We want to seperate it from Point because Point can have different callees and return values on different call events.

Thoughts: Now the parameter for the payload functions has been changed from Point to this PointCallHandle, because the abstraction of Point misses the property of having different callees and return values at the same location in the code. In the past, the utility functions are all static functions, e.g. users call the long ReturnValue(SpPoint*) function to get the return from this call at this point. Would it make more sense to make these utility functions methods of the PointCallHandle class?

How to propel to the exported functions in shared libraies

SPI instruments the plt stubs to propel to the exported functions. The plt stubs have an indirect jump to the actual functions in the shared libraries. Before executing this indirect jump, SPI computes the effective address of it, finds the callee by offset, and instrument the callee.

Most programs now are doing lazy-binding, meaning when an exported function gets called the first time, the GOT entry is not populated yet and a resolver function gets called instead to get the appropriate address of the exported function and jumps to it. To solve this issue, we instrument the resolver function. Because the resolver function eventually jumps to the exported function, we captures that indirect jump will not miss the unbound exported function even this is its first occurrence.

Snippet creation(snippet.cc)

This is the core part of the snippet creation logic. For each and every call instrumentation point, a snippet is constructed. This snippet,

1. Saves registers(including floating point registers and flag regsiters)
2. Calls user's pre-instrumentation function.
3. Restores the saved registers
4. Call the original function
5. Saved registers again
6. Calls user's post-instrumentation function
7. Restore registers back
8. Jump back to the original instruction address.

The following is an constructed snippet for a instrumentation point in a function by Self-propelled.

1. Saves registers by pushing them onto stack

```
55768be95000( 1 bytes): push %rdi               | 57
55768be95001( 1 bytes): push %rsi               | 56
55768be95002( 1 bytes): push %rdx               | 52
55768be95003( 1 bytes): push %rcx               | 51
55768be95004( 2 bytes): push %r8                | 41 50
55768be95006( 2 bytes): push %r9                | 41 51
55768be95008( 1 bytes): push %rax               | 50
55768be95009( 1 bytes): lahf                    | 9f
55768be9500a( 3 bytes): seto %al                |  f 90 c0
55768be9500d( 1 bytes): push %rax               | 50
55768be9500e(10 bytes): mov 93970676425176x,%rax  | 48 b8 d8 a1 c9 40 77 55  0  0
```

```
55768be95018( 3 bytes): mov %rsp,(%rax)           | 48 89 20
55768be9501b( 8 bytes): lea 0xffffff78(%rsp),%rsp | 48 8d a4 24 78 ff ff ff
55768be95023( 3 bytes): mov %rsp,%rax             | 48 8b c4
55768be95026( 6 bytes): add 8x,%rax               | 48  5  8  0  0  0
55768be9502c( 4 bytes): movdqa %xmm0,(%rax)       | 66  f 7f  0
55768be95030( 5 bytes): movdqa %xmm1,0x10(%rax)   | 66  f 7f 48 10
55768be95035( 5 bytes): movdqa %xmm2,0x20(%rax)   | 66  f 7f 50 20
55768be9503a( 5 bytes): movdqa %xmm3,0x30(%rax)   | 66  f 7f 58 30
55768be9503f( 5 bytes): movdqa %xmm4,0x40(%rax)   | 66  f 7f 60 40
55768be95044( 5 bytes): movdqa %xmm5,0x50(%rax)   | 66  f 7f 68 50
55768be95049( 5 bytes): movdqa %xmm6,0x60(%rax)   | 66  f 7f 70 60
55768be9504e( 5 bytes): movdqa %xmm7,0x70(%rax)   | 66  f 7f 78 70
55768be95053( 1 bytes): push %rax                 | 50
```

2. Calls user's precall instrumentation code

```
55768be95054(10 bytes): mov 93970663280928x,%rdi  | 48 bf 20 11  1 40 77 55  0  0
55768be9505e(10 bytes): mov 139922252131528x,%rsi | 48 be c8 6c 22 30 42 7f  0  0
55768be95068(10 bytes): mov 139922228507687x,%r11 | 49 bb 27 f4 b9 2e 42 7f  0  0
55768be95072( 3 bytes): call %r11                 | 41 ff d3
```

3. Restores registers back by popping them from the stack

```
55768be95075( 1 bytes): pop %rax                  | 58
55768be95076( 3 bytes): mov %rsp,%rax             | 48 8b c4
55768be95079( 6 bytes): add 8x,%rax               | 48  5  8  0  0  0
55768be9507f( 4 bytes): movdqa (%rax),%xmm0       | 66  f 6f  0
55768be95083( 5 bytes): movdqa 0x10(%rax),%xmm1   | 66  f 6f 48 10
55768be95088( 5 bytes): movdqa 0x20(%rax),%xmm2   | 66  f 6f 50 20
55768be9508d( 5 bytes): movdqa 0x30(%rax),%xmm3   | 66  f 6f 58 30
55768be95092( 5 bytes): movdqa 0x40(%rax),%xmm4   | 66  f 6f 60 40
55768be95097( 5 bytes): movdqa 0x50(%rax),%xmm5   | 66  f 6f 68 50
55768be9509c( 5 bytes): movdqa 0x60(%rax),%xmm6   | 66  f 6f 70 60
55768be950a1( 5 bytes): movdqa 0x70(%rax),%xmm7   | 66  f 6f 78 70
55768be950a6( 8 bytes): lea 0x88(%rsp),%rsp       | 48 8d a4 24 88  0  0  0
55768be950ae( 1 bytes): pop %rax                  | 58
55768be950af( 3 bytes): add 127x,%al              | 80 c0 7f
55768be950b2( 1 bytes): sahf                      | 9e
55768be950b3( 1 bytes): pop %rax                  | 58
55768be950b4( 2 bytes): pop %r9                   | 41 59
55768be950b6( 2 bytes): pop %r8                   | 41 58
55768be950b8( 1 bytes): pop %rcx                  | 59
55768be950b9( 1 bytes): pop %rdx                  | 5a
55768be950ba( 1 bytes): pop %rsi                  | 5e
55768be950bb( 1 bytes): pop %rdi                  | 5f
```

4. Call the original call instruction

```
55768be950bc( 5 bytes): call 3957976880x          | e8 6f 9e  0 60
```

5. Push the registers back into the stack

```
55768be950c1( 1 bytes): push %rdi                 | 57
55768be950c2( 1 bytes): push %rsi                 | 56
55768be950c3( 1 bytes): push %rdx                 | 52
55768be950c4( 1 bytes): push %rcx                 | 51
55768be950c5( 2 bytes): push %r8                  | 41 50
55768be950c7( 2 bytes): push %r9                  | 41 51
55768be950c9( 1 bytes): push %rax                 | 50
55768be950ca( 1 bytes): lahf                      | 9f
55768be950cb( 3 bytes): seto %al                  |  f 90 c0
55768be950ce( 1 bytes): push %rax                 | 50
55768be950cf(10 bytes): mov 93970676425176x,%rax  | 48 b8 d8 a1 c9 40 77 55  0  0
55768be950d9( 3 bytes): mov %rsp,(%rax)           | 48 89 20
55768be950dc( 8 bytes): lea 0xffffff78(%rsp),%rsp | 48 8d a4 24 78 ff ff ff
55768be950e4( 3 bytes): mov %rsp,%rax             | 48 8b c4
55768be950e7( 6 bytes): add 8x,%rax               | 48  5  8  0  0  0
55768be950ed( 4 bytes): movdqa %xmm0,(%rax)       | 66  f 7f  0
55768be950f1( 5 bytes): movdqa %xmm1,0x10(%rax)   | 66  f 7f 48 10
55768be950f6( 5 bytes): movdqa %xmm2,0x20(%rax)   | 66  f 7f 50 20
55768be950fb( 5 bytes): movdqa %xmm3,0x30(%rax)   | 66  f 7f 58 30
55768be95100( 5 bytes): movdqa %xmm4,0x40(%rax)   | 66  f 7f 60 40
55768be95105( 5 bytes): movdqa %xmm5,0x50(%rax)   | 66  f 7f 68 50
55768be9510a( 5 bytes): movdqa %xmm6,0x60(%rax)   | 66  f 7f 70 60
55768be9510f( 5 bytes): movdqa %xmm7,0x70(%rax)   | 66  f 7f 78 70
55768be95114( 1 bytes): push %rax                 | 50
```

6. Call the user's postcall instrumentation code

```
55768be95115(10 bytes): mov 93970663280928x,%rdi  | 48 bf 20 11  1 40 77 55  0  0
55768be9511f(10 bytes): mov 139922252131629x,%rsi | 48 be 2d 6d 22 30 42 7f  0  0
55768be95129(10 bytes): mov 139922228508586x,%r11 | 49 bb aa f7 b9 2e 42 7f  0  0
55768be95133( 3 bytes): call %r11                 | 41 ff d3
```

7. Restore the registers back by popping them from the stack

8. Jump back

```
399bbd( 5 bytes): jmp 412c96 | e9 d4 90 7 0
```

Snippet Installation(inst_workers/*_worker_impl.*)

To install snippets, we try these instrumentation workers in order.

1. Relocation Call Instruction Worker(inst_workers/callinsn_worker_impl.cc):

This replaces call instruction with a 5 byte jump instruction to the appropriate snippet block created for this point. For eg. this is how a call instruction will be replaced by a jump instruction.

The basic block of the call instruction before installing the snippet:

```
412c78( 1 bytes): push RBP, RSP                       | 55
412c79( 3 bytes): mov RBP, RSP                        | 48 89 e5
412c7c( 4 bytes): sub RSP, 10                         | 48 83 ec 10
412c80( 3 bytes): mov [RBP + fffffffffffffffc], EDI    | 89 7d fc
412c83( 4 bytes): mov [RBP + fffffffffffffff0], RSI    | 48 89 75 f0
412c87( 5 bytes): mov RSI, 2                          | be 2 0 0 0
412c8c( 5 bytes): mov RDI, 41e581                     | bf 81 e5 41 0
412c91( 5 bytes): call 40f400                         | e8 6a c7 ff ff
```

The basic block of the call instruction after installing the snippet:

```
412c78( 1 bytes): push RBP, RSP                     | 55
412c79( 3 bytes): mov RBP, RSP                      | 48 89 e5
412c7c( 4 bytes): sub RSP, 10                       | 48 83 ec 10
412c80( 3 bytes): mov [RBP + fffffffffffffffc], EDI | 89 7d fc
412c83( 4 bytes): mov [RBP + fffffffffffffff0], RSI | 48 89 75 f0
412c87( 5 bytes): mov RSI, 2                        | be 2 0 0 0
412c8c( 5 bytes): mov RDI, 41e581                   | bf 81 e5 41 0
412c91( 5 bytes): jmp 399a00                        | e9 6a 6d f8 ff
```

Here note that call instruction at 412c91 is replaced by jump instruction. The address in the jump instruction '399a00' is the relative location of the snippet from 412c91. If the call instruction is less than 5 bytes, this worker will fail to install a snippet. Then the subsequent workers will be try to install the snippet.

2. Relocation Call Block Worker(inst_workers/callinsn_worker_impl.cc):

This worker replaces the basic block associated with the call instruction with a jump instruction.

The instructions which are in the basic block above the call instruction will be relocated to the starting address of the snippet.

The basic block of the call instruction before installing the snippet:

```
7f241c4aa229( 7 bytes): mov RAX, [RIP + 327b18]   | 48 8b 5 18 7b 32 0
7f241c4aa230( 3 bytes): mov RCX, [RAX]            | 48 8b 8
7f241c4aa233( 7 bytes): mov RDX, [RBP + fffffe60] | 48 8b 95 60 fe ff ff
7f241c4aa23a( 6 bytes): mov EAX, [RBP + fffffe6c] | 8b 85 6c fe ff ff
7f241c4aa240( 3 bytes): mov RSI, RDX              | 48 89 d6
7f241c4aa243( 2 bytes): mov EDI, EAX              | 89 c7
7f241c4aa245( 2 bytes): call RCX                  | ff d1
```

The basic block of the call instruction after installing the snippet:

```
7f241c4aa229( 5 bytes): jmp ac9cb00 | e9 d2 28 7f ee
```

The instructions

```
7f241c4aa229( 7 bytes): mov RAX, [RIP + 327b18] | 48 8b 5 18 7b 32 0
```

```
7f241c4aa230( 3 bytes): mov RCX, [RAX] | 48 8b 8
7f241c4aa233( 7 bytes): mov RDX, [RBP + fffffe60] | 48 8b 95 60 fe ff ff
7f241c4aa23a( 6 bytes): mov EAX, [RBP + fffffe6c] | 8b 85 6c fe ff ff
7f241c4aa240( 3 bytes): mov RSI, RDX | 48 89 d6
7f241c4aa243( 2 bytes): mov EDI, EAX | 89 c7
```

which occurs before the call instructions will be relocated to the starting address of the snippet.

This may fail if the jump instruction is larger than a call block.

3. Spring Board Worker(inst_workers/spring_worker_impl.cc):

Relocates call block C and replaces the call block with a short jump that transfers control to a

nearby block as a spring board block S. Relocates S and replace S wih two jump instructions,

one of which jumps to the relocated S(denoted as S‘) and the other jumps to the snippet. To

sum up, C>S>S’>S>snippet>C. This may fail if a nearby springboard large wnouh to place

two long jumps can not be found.

4. Trap Worker(inst_workers/trap_worker_impl.cc):

Relocates call instruction and replaces with a jump instruction.

The basic block of the call instruction before installing trap:

```
7f97ba85c145( 3 bytes): mov RAX, RBX | 48 89 d8
7f97ba85c148( 3 bytes): mov RDI, RAX | 48 89 c7
7f97ba85c14b( 5 bytes): call ba81f890 | e8 40 37 fc ff
```

The basic block of the call instruction after installing trap

```
7f97ba85c145( 3 bytes): mov RAX, RBX | 48 89 d8
7f97ba85c148( 3 bytes): mov RDI, RAX | 48 89 c7
7f97ba85c14b( 1 bytes): int 3 | cc
```

The trap handler transfers control to the appropriate snippet when the trap instruction is invoked.

Interprocess propagation: Refer SPI documentation

SpAddrSpace(patchapi/addr_space.cc) and SpObject(patchapi/object-x86_64.cc)

These two classes are from PatchAPI interface. SpAddrSpace finds the available free intervals from the gaps in between the code objects and splits them into small free intervals. It then binds a free interval to each code object that will be instrumented(Note that a code object will not get a free interval if it is not being instrumented). When a SpObject runs out of memory in its free interval, it asks SpAddrSpace to for a new free interval.

Note: Future Improvement (github issue #8 https://github.com/dyninst/spi/issues/8) Sometimes we run out of free intervals when doing instrumentation. When the SpAddrSpace binds a free interval to a SpObject, there are two constrains: 1. The interval has to be within 1.5GB of the SpObject, 2. The interval has to be before the SpObject, meaning the address of the interval has to be smaller than the SpObject. I don't fully understand the reasons behind these two constrains. The first constrain might due to the jump offset limit. The second is possibly just for implementation simplicity. If the second constrain is not necessary, then free interval binding process will be much more flexible and we are less likely to run out of free intervals.

What are the plugins

SPI is designed to be an infrastructure for doing instrumentations. We allow users to simply write few high level C++ functions to do function-level instrumentation. When users write their own payload functions and build, they generate their own agent library with the dependency on our libagent.so. **The plugins** are basically user agent library provided by us. For example, the FPVA plugin is a user agent library designed for doing security analysis. The payload functions are complex enough to become a project on its own. The plugins can also showcase how to use SPI.