This document summarizes my understanding of how Self-propelled works.

libmyagent.so: Its init code registers user specified pre-instrumentation and post instrumentation and calls Agent's *Go()* function. Init code will be executed when this shared library will be injected/LD_PRELOADed when the program runs.

Agent:(agent.cc)

a. Initializes the default entries if user-instrumentation code is not available

b. Parses the binary using ParseAPI and PatchAPI and records the necessary data structures(parser.cc)

c. Sets up configuration parameters such as payload entry function, payload exit function and other necessary things.

d. Registers event

Event(event.cc)

a. Determines if a function is injected/ preloaded by looking at the call stack size. If the call stack size if 0, the agent is considered preloaded, otherwise, it is considered to be injected.

b. If it is found to be preloaded, then it will find *main()* function, and will propel(propeller.cc) inside that. Otherwise, it will propel into all the function present in the call stack until it finds main.

Propeller (propeller.cc)

a. Finds instrumentation points to instrument inside the given function(handles both direct and indirect call instructions)

b. Calls patcher.commit which internally calls run() function, which is defined in Instrumenter

Instrumenter(patchapi/instrumenter.cc)

a. For each of the instrumentation points, instrumenter creates and installs snippet. Snippet creation is done at snippet.cc and installation will be done by different workers.

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-call 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

```
399a00( 1 bytes): push RDI, RSP            | 57
399a01( 1 bytes): push RSI, RSP            | 56
399a02( 1 bytes): push RDX, RSP            | 52
399a03( 1 bytes): push RCX, RSP            | 51
399a04( 2 bytes): push R8, RSP             | 41 50
399a06( 2 bytes): push R9, RSP             | 41 51
399a08( 1 bytes): push RAX, RSP            | 50
399a09( 1 bytes): lahf                     | 9f
399a0a( 3 bytes): seto AL                  |  f 90 c0
399a0d( 1 bytes): push RAX, RSP            | 50
399a0e(10 bytes): mov RAX, 1df1d398        | 48 b8 98 d3 f1 1d  0  0  0  0
399a18( 3 bytes): mov [RAX], RSP           | 48 89 20
399a1b( 2 bytes): push R10, RSP            | 41 52
399a1d( 2 bytes): push R11, RSP            | 41 53
399a1f( 1 bytes): push RBX, RSP            | 53
399a20( 2 bytes): push R12, RSP            | 41 54
399a22( 2 bytes): push R13, RSP            | 41 55
399a24( 2 bytes): push R14, RSP            | 41 56
399a26( 2 bytes): push R15, RSP            | 41 57
399a28( 1 bytes): push RBP, RSP            | 55
399a29( 8 bytes): lea RSP, ESP + ffffff78  | 48 8d a4 24 78 ff ff ff
399a31( 3 bytes): mov RAX, RSP             | 48 8b c4
399a34( 6 bytes): add RAX, 8               | 48  5  8  0  0  0
399a3a( 4 bytes): movdqa [RAX], XMM0       | 66  f 7f  0
399a3e( 5 bytes): movdqa [RAX + 10], XMM1  | 66  f 7f 48 10
399a43( 5 bytes): movdqa [RAX + 20], XMM2  | 66  f 7f 50 20
399a48( 5 bytes): movdqa [RAX + 30], XMM3  | 66  f 7f 58 30
399a4d( 5 bytes): movdqa [RAX + 40], XMM4  | 66  f 7f 60 40
399a52( 5 bytes): movdqa [RAX + 50], XMM5  | 66  f 7f 68 50
399a57( 5 bytes): movdqa [RAX + 60], XMM6  | 66  f 7f 70 60
399a5c( 5 bytes): movdqa [RAX + 70], XMM7  | 66  f 7f 78 70
399a61( 1 bytes): push RAX, RSP            | 50
```

//2. Calls user’s pre-call instrumentation code

```
399a62(10 bytes): mov RDI, 1df1cf20        | 48 bf 20 cf f1 1d  0  0  0  0
399a6c(10 bytes): mov RSI, 1c8a1dac        | 48 be ac 1d 8a 1c 24 7f  0  0
399a76(10 bytes): mov R11, 1ae47e04        | 49 bb  4 7e e4 1a 24 7f  0  0
399a80( 3 bytes): call R11                 | 41 ff d3
```

//3. Restores registers back by popping them from the stack

```
399a83( 1 bytes): pop RAX, RSP             | 58
399a84( 3 bytes): mov RAX, RSP             | 48 8b c4
399a87( 6 bytes): add RAX, 8               | 48  5  8  0  0  0
399a8d( 4 bytes): movdqa XMM0, [RAX]       | 66  f 6f  0
399a91( 5 bytes): movdqa XMM1, [RAX + 10]  | 66  f 6f 48 10
399a96( 5 bytes): movdqa XMM2, [RAX + 20]  | 66  f 6f 50 20
399a9b( 5 bytes): movdqa XMM3, [RAX + 30]  | 66  f 6f 58 30
399aa0( 5 bytes): movdqa XMM4, [RAX + 40]  | 66  f 6f 60 40
399aa5( 5 bytes): movdqa XMM5, [RAX + 50]  | 66  f 6f 68 50
399aaa( 5 bytes): movdqa XMM6, [RAX + 60]  | 66  f 6f 70 60
399aaf( 5 bytes): movdqa XMM7, [RAX + 70]  | 66  f 6f 78 70
399ab4( 8 bytes): lea RSP, ESP + 88        | 48 8d a4 24 88  0  0  0
399abc( 1 bytes): pop RBP, RSP             | 5d
399abd( 2 bytes): pop R15, RSP             | 41 5f
399abf( 2 bytes): pop R14, RSP             | 41 5e
```

```
399ac1( 2 bytes): pop R13, RSP              | 41 5d
399ac3( 2 bytes): pop R12, RSP              | 41 5c
399ac5( 1 bytes): pop RBX, RSP              | 5b
399ac6( 2 bytes): pop R11, RSP              | 41 5b
399ac8( 2 bytes): pop R10, RSP              | 41 5a
399aca( 1 bytes): pop RAX, RSP              | 58
399acb( 3 bytes): add AL, 7f                | 80 c0 7f
399ace( 1 bytes): sahf                      | 9e
399acf( 1 bytes): pop RAX, RSP              | 58
399ad0( 2 bytes): pop R9, RSP               | 41 59
399ad2( 2 bytes): pop R8, RSP               | 41 58
399ad4( 1 bytes): pop RCX, RSP              | 59
399ad5( 1 bytes): pop RDX, RSP              | 5a
399ad6( 1 bytes): pop RSI, RSP              | 5e
399ad7( 1 bytes): pop RDI, RSP              | 5f
```

//4. Call the original call instruction

```
399ad8(10 bytes): mov R11, 1c37ca39         | 49 bb 39 ca 37 1c 24 7f  0  0
399ae2( 3 bytes): call R11                  | 41 ff d3
```

//5. Push the registers back into the stack

```
399ae5( 1 bytes): push RDI, RSP             | 57
399ae6( 1 bytes): push RSI, RSP             | 56
399ae7( 1 bytes): push RDX, RSP             | 52
399ae8( 1 bytes): push RCX, RSP             | 51
399ae9( 2 bytes): push R8, RSP              | 41 50
399aeb( 2 bytes): push R9, RSP              | 41 51
399aed( 1 bytes): push RAX, RSP             | 50
399aee( 1 bytes): lahf                      | 9f
399aef( 3 bytes): seto AL                   |  f 90 c0
399af2( 1 bytes): push RAX, RSP             | 50
399af3(10 bytes): mov RAX, 1df1d398         | 48 b8 98 d3 f1 1d  0  0  0  0
399afd( 3 bytes): mov [RAX], RSP            | 48 89 20
399b00( 2 bytes): push R10, RSP             | 41 52
399b02( 2 bytes): push R11, RSP             | 41 53
399b04( 1 bytes): push RBX, RSP             | 53
399b05( 2 bytes): push R12, RSP             | 41 54
399b07( 2 bytes): push R13, RSP             | 41 55
399b09( 2 bytes): push R14, RSP             | 41 56
399b0b( 2 bytes): push R15, RSP             | 41 57
399b0d( 1 bytes): push RBP, RSP             | 55
399b0e( 8 bytes): lea RSP, ESP + ffffff78   | 48 8d a4 24 78 ff ff ff
399b16( 3 bytes): mov RAX, RSP              | 48 8b c4
399b19( 6 bytes): add RAX, 8                | 48  5  8  0  0  0
399b1f( 4 bytes): movdqa [RAX], XMM0        | 66  f 7f  0
399b23( 5 bytes): movdqa [RAX + 10], XMM1   | 66  f 7f 48 10
399b28( 5 bytes): movdqa [RAX + 20], XMM2   | 66  f 7f 50 20
399b2d( 5 bytes): movdqa [RAX + 30], XMM3   | 66  f 7f 58 30
399b32( 5 bytes): movdqa [RAX + 40], XMM4   | 66  f 7f 60 40
399b37( 5 bytes): movdqa [RAX + 50], XMM5   | 66  f 7f 68 50
399b3c( 5 bytes): movdqa [RAX + 60], XMM6   | 66  f 7f 70 60
399b41( 5 bytes): movdqa [RAX + 70], XMM7   | 66  f 7f 78 70
399b46( 1 bytes): push RAX, RSP             | 50
```

//6. Call the user’s post-call instrumentation code

```
399b47(10 bytes): mov RDI, 1df1cf20         | 48 bf 20 cf f1 1d  0  0  0  0
```

```
399b51(10 bytes): mov RSI, 1c8a1dfa        | 48 be fa 1d 8a 1c 24 7f  0  0
399b5b(10 bytes): mov R11, 1ae47eb0        | 49 bb b0 7e e4 1a 24 7f  0  0
399b65( 3 bytes): call R11                 | 41 ff d3
```

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

```
399b68( 1 bytes): pop RAX, RSP             | 58
399b69( 3 bytes): mov RAX, RSP             | 48 8b c4
399b6c( 6 bytes): add RAX, 8               | 48  5  8  0  0  0
399b72( 4 bytes): movdqa XMM0, [RAX]       | 66  f 6f  0
399b76( 5 bytes): movdqa XMM1, [RAX + 10]  | 66  f 6f 48 10
399b7b( 5 bytes): movdqa XMM2, [RAX + 20]  | 66  f 6f 50 20
399b80( 5 bytes): movdqa XMM3, [RAX + 30]  | 66  f 6f 58 30
399b85( 5 bytes): movdqa XMM4, [RAX + 40]  | 66  f 6f 60 40
399b8a( 5 bytes): movdqa XMM5, [RAX + 50]  | 66  f 6f 68 50
399b8f( 5 bytes): movdqa XMM6, [RAX + 60]  | 66  f 6f 70 60
399b94( 5 bytes): movdqa XMM7, [RAX + 70]  | 66  f 6f 78 70
399b99( 8 bytes): lea RSP, ESP + 88        | 48 8d a4 24 88  0  0  0
399ba1( 1 bytes): pop RBP, RSP             | 5d
399ba2( 2 bytes): pop R15, RSP             | 41 5f
399ba4( 2 bytes): pop R14, RSP             | 41 5e
399ba6( 2 bytes): pop R13, RSP             | 41 5d
399ba8( 2 bytes): pop R12, RSP             | 41 5c
399baa( 1 bytes): pop RBX, RSP             | 5b
399bab( 2 bytes): pop R11, RSP             | 41 5b
399bad( 2 bytes): pop R10, RSP             | 41 5a
399baf( 1 bytes): pop RAX, RSP             | 58
399bb0( 3 bytes): add AL, 7f               | 80 c0 7f
399bb3( 1 bytes): sahf                     | 9e
399bb4( 1 bytes): pop RAX, RSP             | 58
399bb5( 2 bytes): pop R9, RSP              | 41 59
399bb7( 2 bytes): pop R8, RSP              | 41 58
399bb9( 1 bytes): pop RCX, RSP             | 59
399bba( 1 bytes): pop RDX, RSP             | 5a
399bbb( 1 bytes): pop RSI, RSP             | 5e
399bbc( 1 bytes): pop RDI, RSP             | 5f
```

//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.

Inter-process propagation: Refer SPI documentation

Other miscellaneous stuff learnt

nm <<object_name>> - lists the symbols in the object file. eg. nm libagent.so
objdump -d <<object_name>> - dumps the object code with symbols.
ldd <<object_name>> - lists the dynamic linked libraries
c++filt <<mangled_symbol>> - gives the original function in the source code

Dependency issues in Dynisnt can be solved by
find . -name DEPENDS | xargs rm

If error is encountered during Dyninst compilation, do the following:

```
find . -name DEPENDS | xargs rm
make distclean
./configure --with-package-base=/p/paradyn/packages
make -j8
```

```
./configure --with-package-base=/p/paradyn/packages --prefix=${DYNINST_ROOT}
--exec-prefix=${DYNINST_ROOT}/${PLATFORM} --disable-testsuite
make install
```

Sometimes, pushing and popping items(when storing/restoring regsiters) from the stack leave it unaligned. In x86- 64, the stack should be 16 byte aligned before/after every function call, so should be careful when pushing and popping items from the stack.

Sometimes gdb output will be different from the normal run output, this is possible and it depends entirely on the program. To get away from that problem, attach pid instead of launching the program with gdb

LD_PRELOAD of SPI might not work because of permission reasons, check if both the program, the files it is gonna open are having the appropriate permissions.

Things learnt on OOB

Whenever a receiving end receives a OOB packet, it send a SIG_URG signal. (A process or process group can be configured to receive SIGURG signals when out-of-band data is available for reading on a socket, by using the F_SETOWN command of the fcntl() system call- from wiki) . after registering a system call for SIG_URG signal, you can receive an out-of band packet in the signal handler which you have written .

Potential problem:
If an application by itself has registered a sigurg signal handler, then things will be messed up.

Send an auto generated pseudorandom sequence number before every send. Try to receive the pseudo random sequence number before every receive- try to match it with the corresponding send from the host
- try to piggyback on the same send, and check if it is possible.

Problems in sending a sequence number before every send:
Recv has to receive a sequence number which send() sent. On the receiver side, if it is a while loop for recv, then recv() will expect a sequence number before every recv. Also, if something is not instrumented and it gets a IPC recv function, it will first try to get a sequence number, which will not be there unless it is sent by the other function, since IPC accept is also treated as a recv function, this might create a problem.

Time overhead:

1. Flag regsiters are not saved- resulted in the same time, no reduction

Takes 1 minute if just central manager and execute host are instrumented.
Takes 6 seconds if only central manager is instrumented.
Takes 2 minutes if only central manager and submit host are instrumented.

Takes 2.30 - 3 minutes if all of them are instrumented.

I am guessing the startup takes most of them time, since after a condor_submit, schedd forks condor_shadow, and startd forks starter.