Dynamic Userspace Tracing

Tracing Summit 2016

Embedded Linux Europe

Presenter Information
François Tétreault and Jason Puncher

Date
October 12, 2016

Dynamic Userspace Tracing

Tracing Summit 2016

Agenda/Key Takeaways

1	**Static Tracing Technique used on Linux with LTTng** • Based on the "-finstrument-functions" GNU compiler option and the "LD_PRELOAD" environment variable • Non architecture specific

2	**Dynamic Tracing techniques used on Linux with LTTng**
	• A 1st approach based on ptrace, mmap, and dynamically modified code
	  • Architecture covered: PowerPC (Ciena's implementation also supports x86 ***)
	• A 2nd approach based on the DyninstAPI library
	  • Architecture supported by Dyninst

3	**Dynamic Tracing technique used on VxWorks** • Based on modified assembly code and processor interrupt exceptions • Architecture covered: PowerPC (Ciena's implementation also supports x86)
4	**Dynamic Instrumentation using GDB with LTTng**
5	**Comparative Slide**

(***) For details about the x86 implementation, you can refer to the previous previous version of this presentation

Overview

Goal of the techniques discussed in this presentation is to dynamically instrument the code

"Doing it live on a binary loaded into RAM"

Ultimately to trace all function calls

The stack can then be viewed in a flame graph

(see TraceCompass Flame Graph View)

Our debug targets:

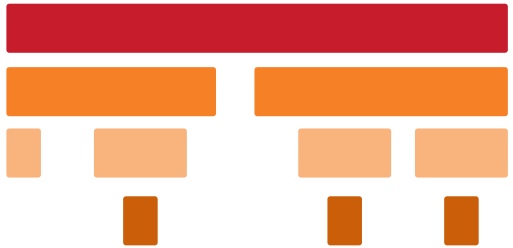
Most are PowerPC-based

File system is either a small flash disk or a tiny ROM

RAM is limited

i.e. some targets only have 64 MB of RAM

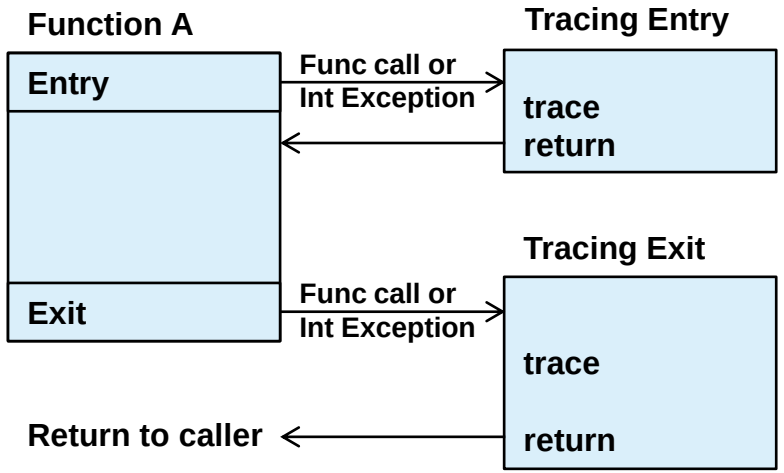
Flame Graph



Overview

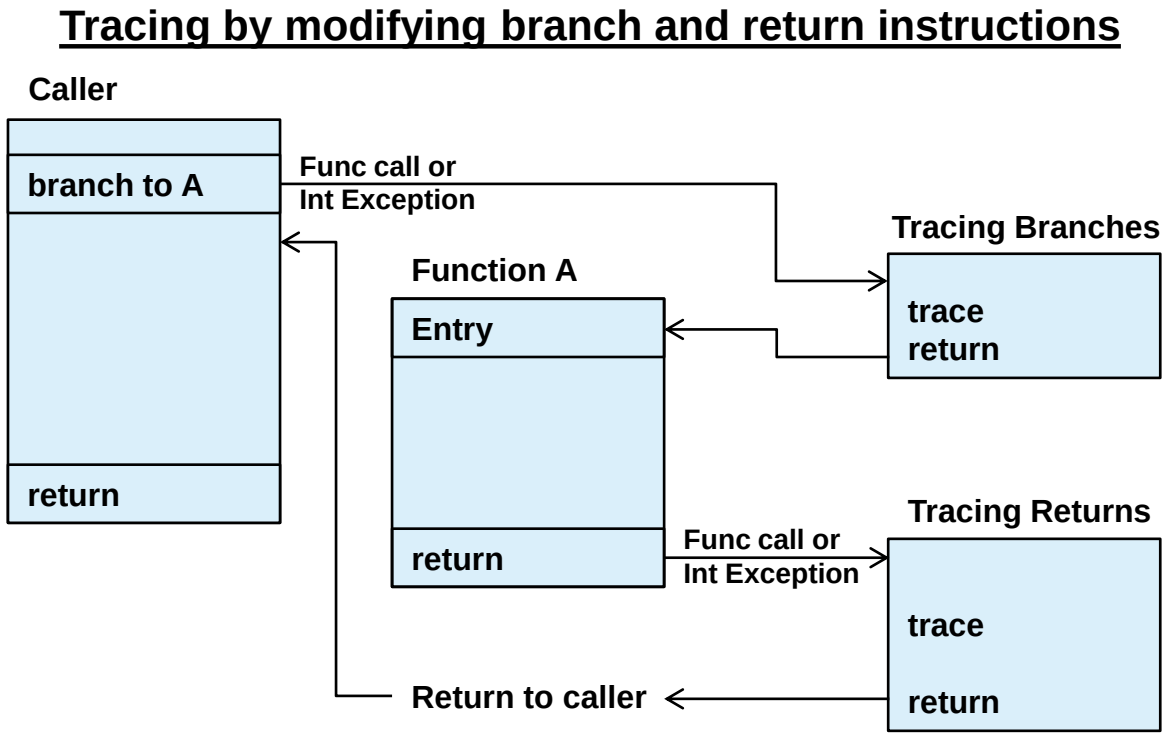
- **Two general approaches are covered in this presentation**
  1. Modifying the entry and exit of functions
     - May either rely on function calls, or
     - interrupt exceptions

Tracing by modifying function entry and exit



Overview

- **Two general approaches are covered in this presentation**
  1. Modifying the entry and exit of functions
     - May either rely on function calls, or
     - interrupt exceptions
  2. Modifying branch instructions and return instructions
     - May either rely on function calls, or
     - interrupt exceptions
     - Particularly useful when the symbol information is not available



Static Tracing Technique used on Linux with LTTng

Static Tracing Technique used on Linux with LTTng

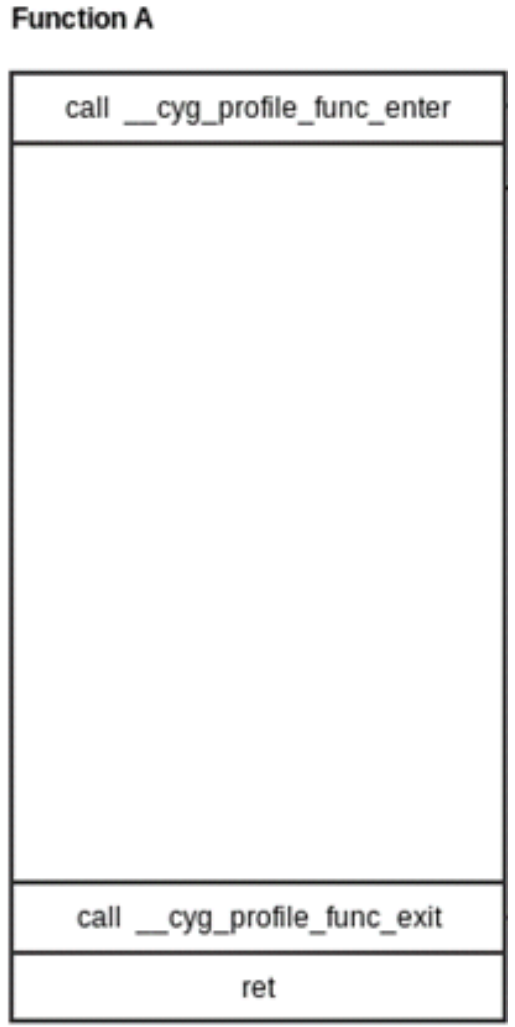
This technique is readily available

It inspired our dynamic techniques used on Linux with LTTng

The GNU compiler has a compile option "-finstrument-functions" which generates instrumentation calls for the entry and exit functions

On every entry into a function, and just before returning from a function the following routines will be called:

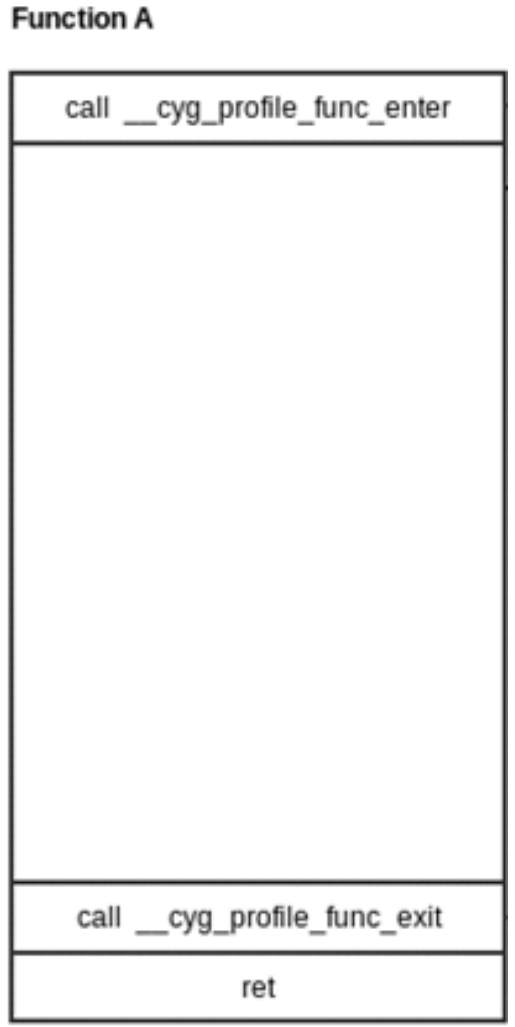
```
void __cyg_profile_func_enter (void *this_fn, void *call_site);

void __cyg_profile_func_exit  (void *this_fn, void *call_site);
```



Static Tracing Technique used on Linux with LTTng

Each routine will be provided with:

- **`this_fn`** → starting address of the function that is being executed (the “callee”)
- **`call_site`** → address from which the function was called (the “caller”)

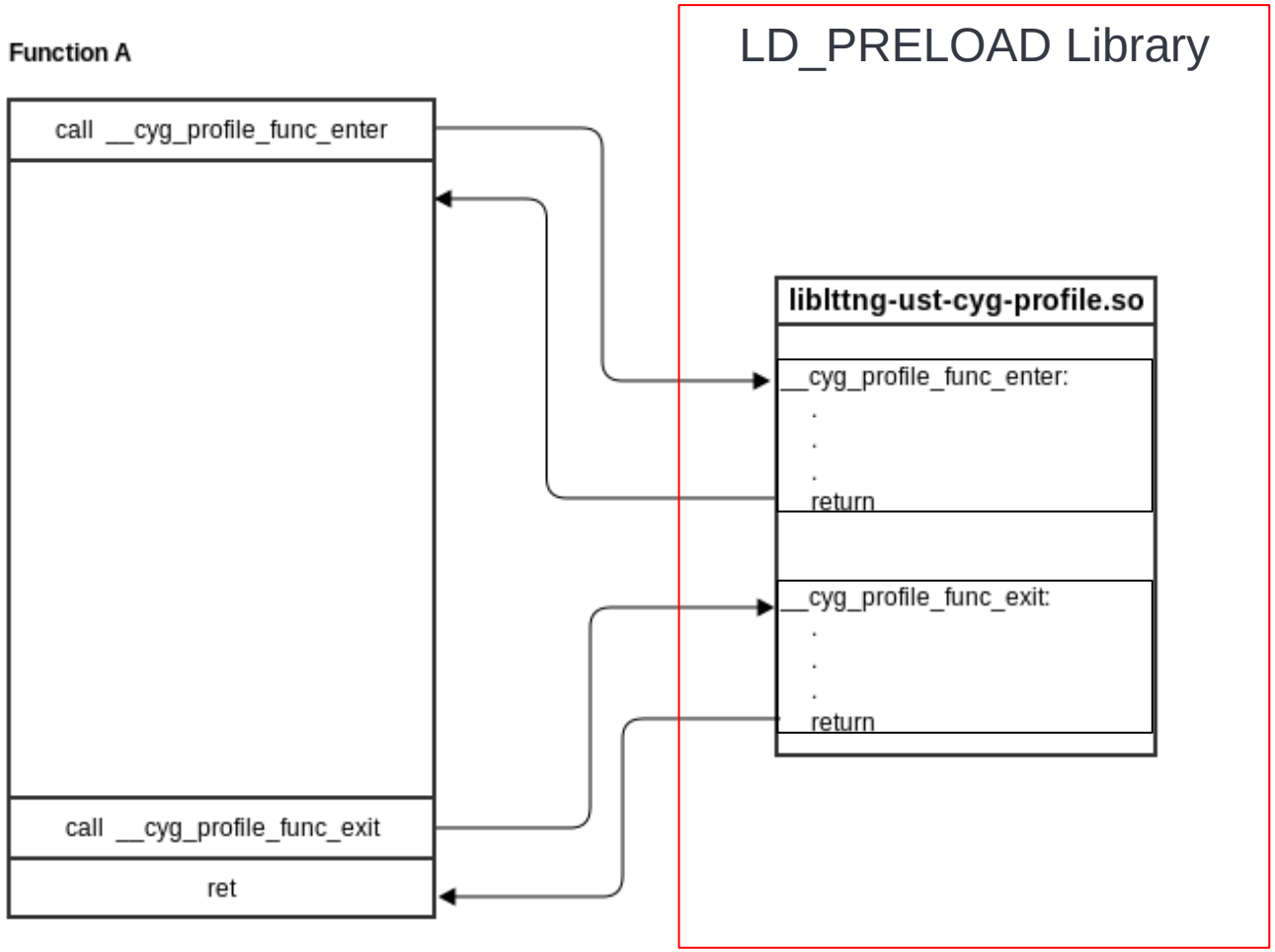


Static Tracing Technique used on Linux with LTTng
→ Static Function Instrumentation Flow

Any shared library function can be replaced using the LD_PRELOAD environment variable

Here we load the liblttng-ust-cyg-profile.so shared library

- LD_PRELOAD allows us to link this library in place of the original function stubs



Static Tracing Technique used on Linux with LTTng

These functions would then log the function entry and exit in LTTng

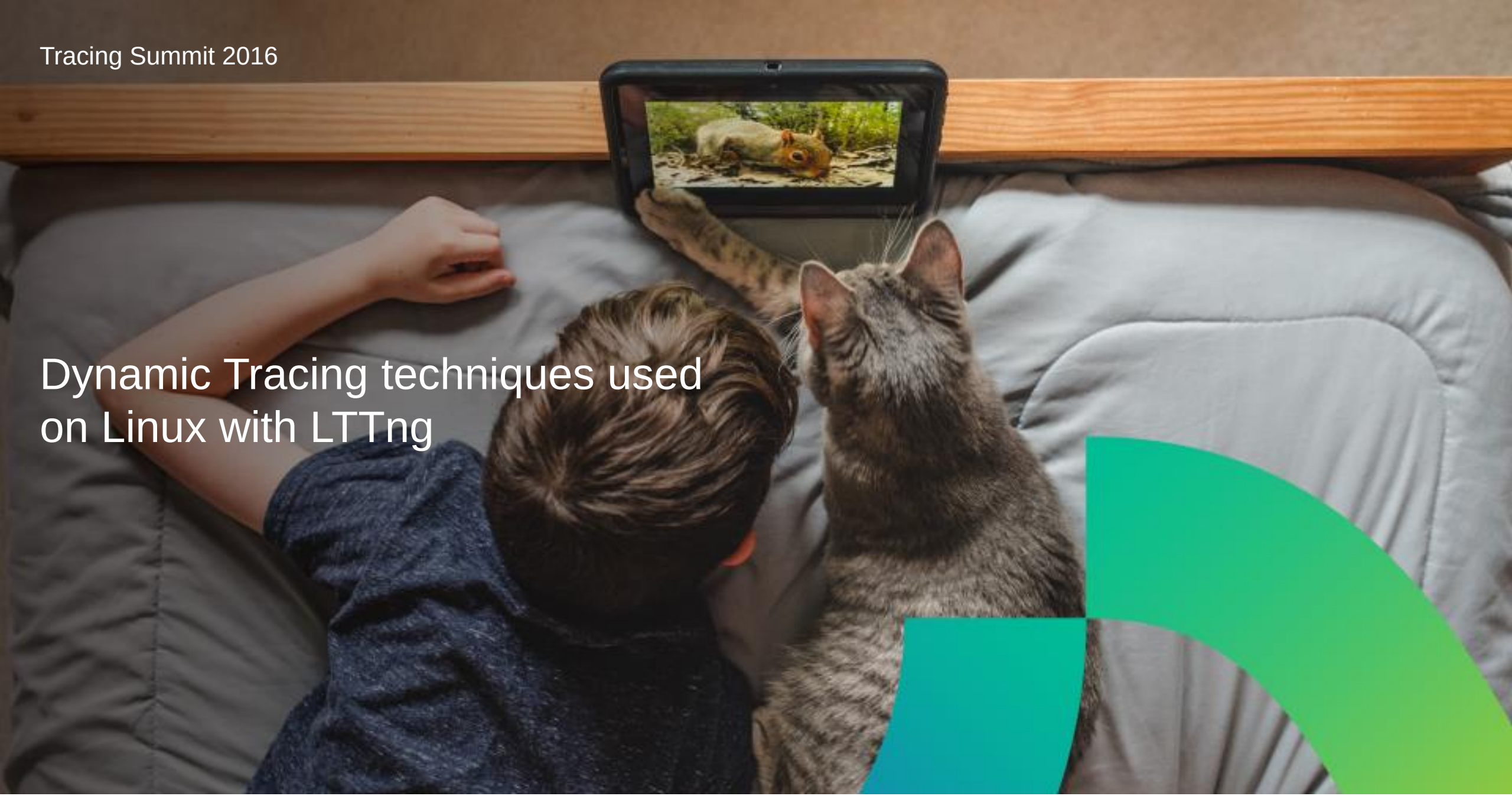
Here's a sample trace:

```
[18:47:20.956030535] (+?.??????????) (none) lttng_ust_cyg_profile:func_entry: { cpu_id = 0 }, { vtid =
3586, procname = "bbb_Upgrade" }, { addr = 0x…, call_site = 0x… }

[18:47:20.956104596] (+0.000074061) (none) lttng_ust_cyg_profile:func_entry: { cpu_id = 0 }, { vtid =
3586, procname = "bbb_Upgrade" }, { addr = 0x…, call_site = 0x… }

[18:47:20.956124114] (+0.000019518) (none) lttng_ust_cyg_profile:func_exit: { cpu_id = 0 }, { vtid =
3586, procname = "bbb_Upgrade" }, { addr = 0x…, call_site = 0x… }

[18:47:20.956130907] (+0.000006793) (none) lttng_ust_cyg_profile:func_entry: { cpu_id = 0 }, { vtid =
3586, procname = "bbb_Upgrade" }, { addr = 0x…, call_site = 0x… }
```

Static Tracing Technique used on Linux with LTTng

This is great! But it has its limitations:

- requires a recompile of the code to insert the calls to the generated entry/exit functions
- logging will be done on all threads within a process
- lacks ability to insert trigger points on which to commence tracing (tracing will start when "lttng start" is called)



Dynamic Tracing techniques used on Linux with LTTng

Dynamic Tracing technique on Linux

The two approaches discussed in this section provide the ability to:

- dynamically instrument the code
  - Insert tracepoints to the entry and exit of each traced routines
  - No recompilation or use of any compile time flags required
- select individual threads on which to capture function entry/exit events
- trigger the start/stop of the trace collection when a particular events occur

Dynamic Tracing technique on Linux

Architecture covered: PowerPC

Ciena also supports x86 for both methods

(x86 is not explained in this presentation)

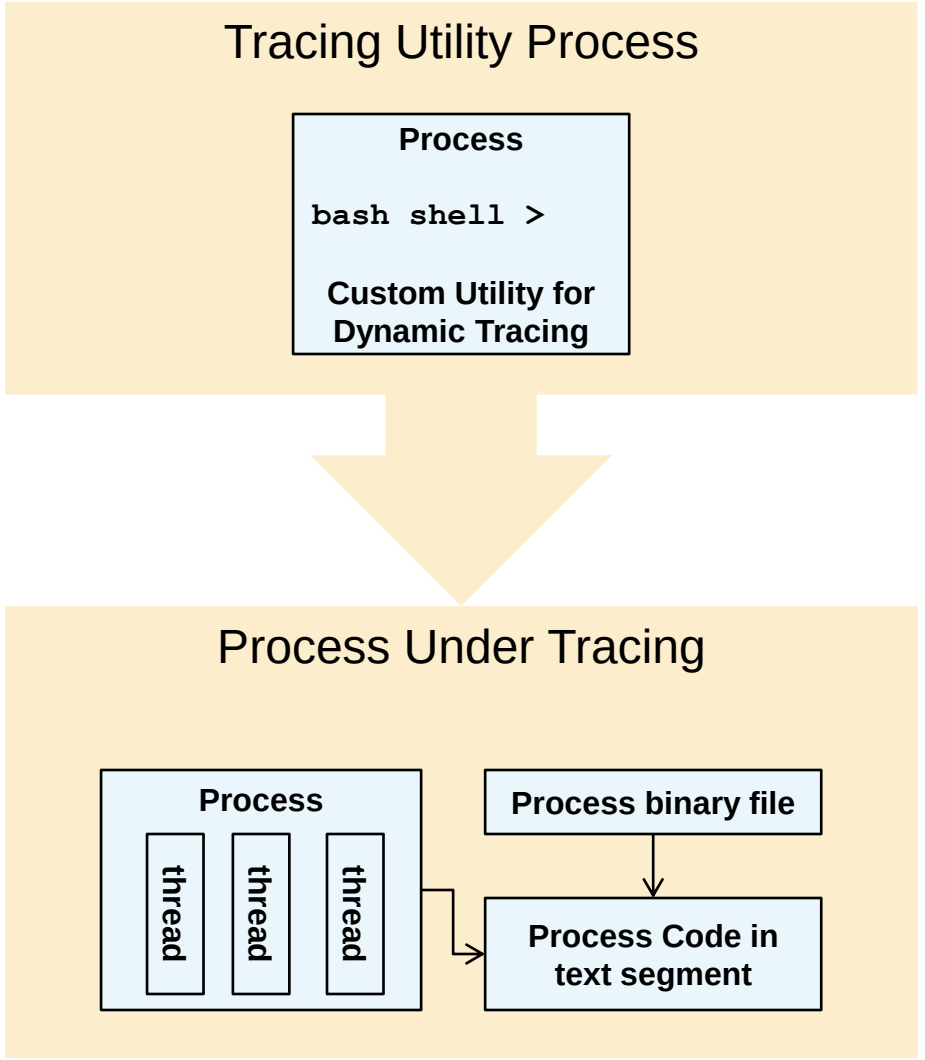
Both methods are implemented and used at Ciena

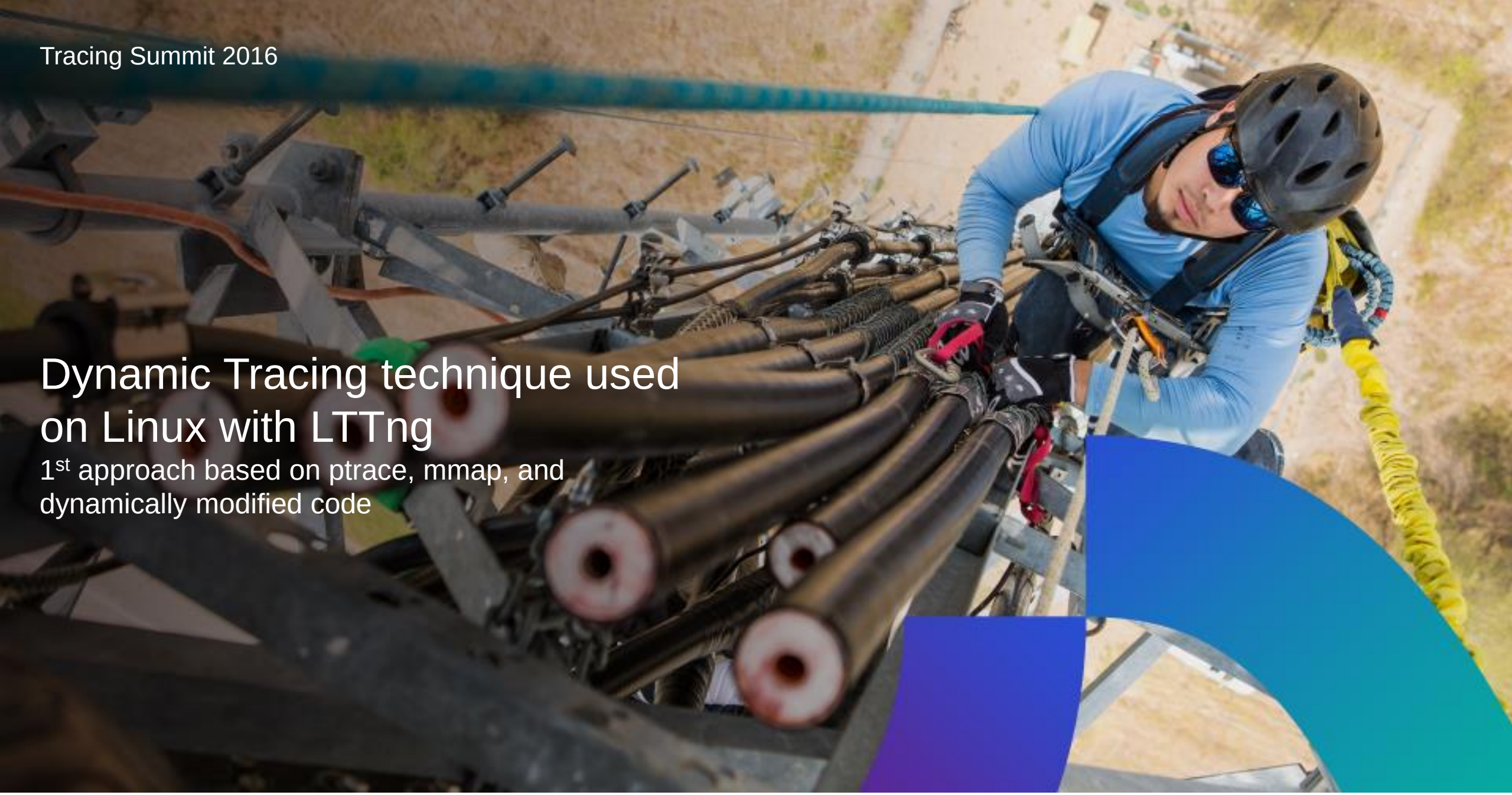
Allows tracing with LTTng in a embedded environment

Context:

- **Executable binary file**
- **Process running the program**
- **Program loaded in RAM**
- **Tracing utility running from a bash shell process**

- **Goal is to trace a target process from the context of the tracing utility**





Dynamic Tracing technique used on Linux with LTTng

1st approach based on ptrace, mmap, and dynamically modified code

Dynamic Tracing technique on Linux with ptrace
→ Dynamic Function Instrumentation Flow

Function A → Function to trace

Tracing is done using liblttng-ust-cyg-profile.so:

- cyg_profile_func_enter
- cyg_profile_func_exit

An entry tracing function is needed to call trace_ust_func_enter and pass:

- `this_fn` →address is hard coded in prologue
  (one prologue per traced function)
- `call_site` → LR (Link Register)

The first instruction is modified to call the entry tracing function

The Epilogue contains the return addr to A and the overwritten instruction (1str instruction)

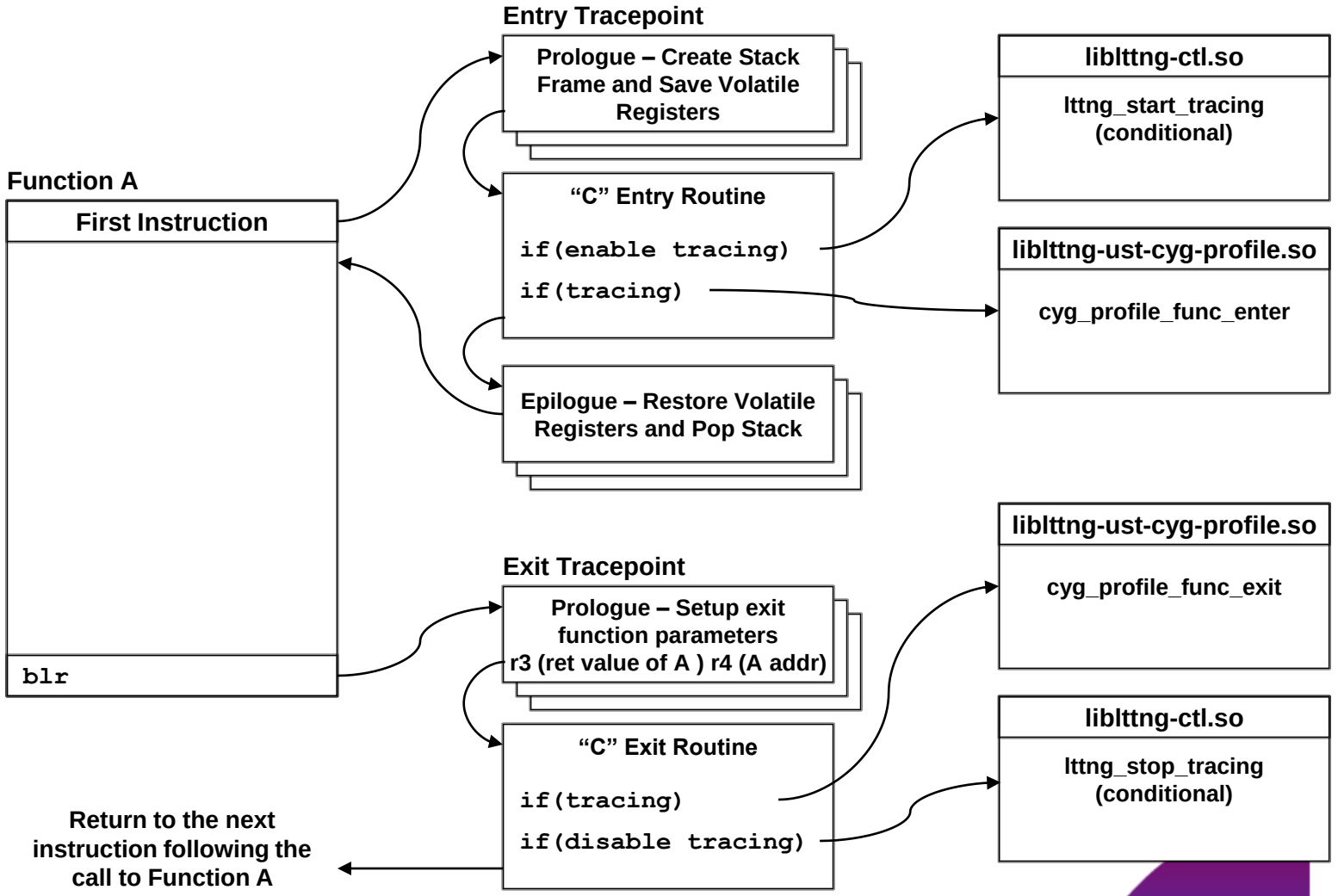
An exit tracing function is needed to call trace_ust_func_exit and pass:

- `this_fn` → address is hard coded in prologue
- `call_site` →LR (Link Register)

The return instruction is modified to call the exit tracing function

After which the flow is returned to the caller

Additional control is added to enable / disable tracing on demand



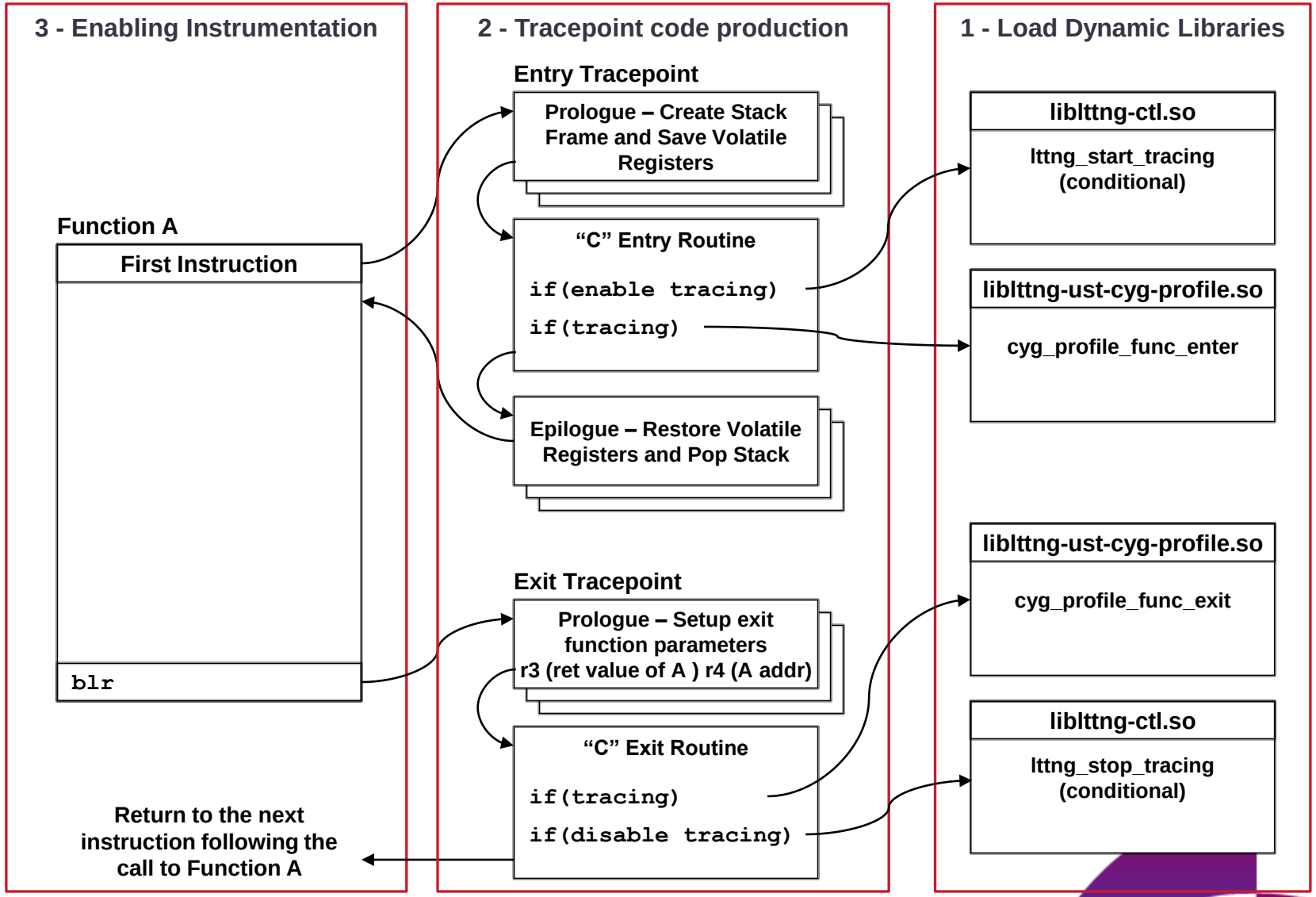
Dynamic Tracing technique on Linux with ptrace
→ Dynamic Function Instrumentation Flow

How is this autonomously setup?

1st stage → load the dynamic libraries

2nd stage → produce the tracepoint code

3rd stage → enable the instrumentation



Dynamic Tracing technique on Linux with ptrace
→ Technology needed in stages 1 to 3

Halt thread execution in order to modify its source

Make inter-process procedure calls in order to call :

- dlopen : to load shared libraries
- dlsym : to lookup symbol addresses (in shared libraries)
- mmap : to allocate memory (in which to copy tracepoints)
- etc…

Modify process memory in order to :

- copy the tracepoint code into the processes memory map
- modify the source in order to insert branches to the newly introduced tracepoints

This is accomplished using ptrace.

Dynamic Tracing technique on Linux with ptrace
→ ptrace (used in all 3 stages)

The ptrace() system call provides a means by which to observe and control the execution of another process. With it you can change the image, and process registers.

```
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

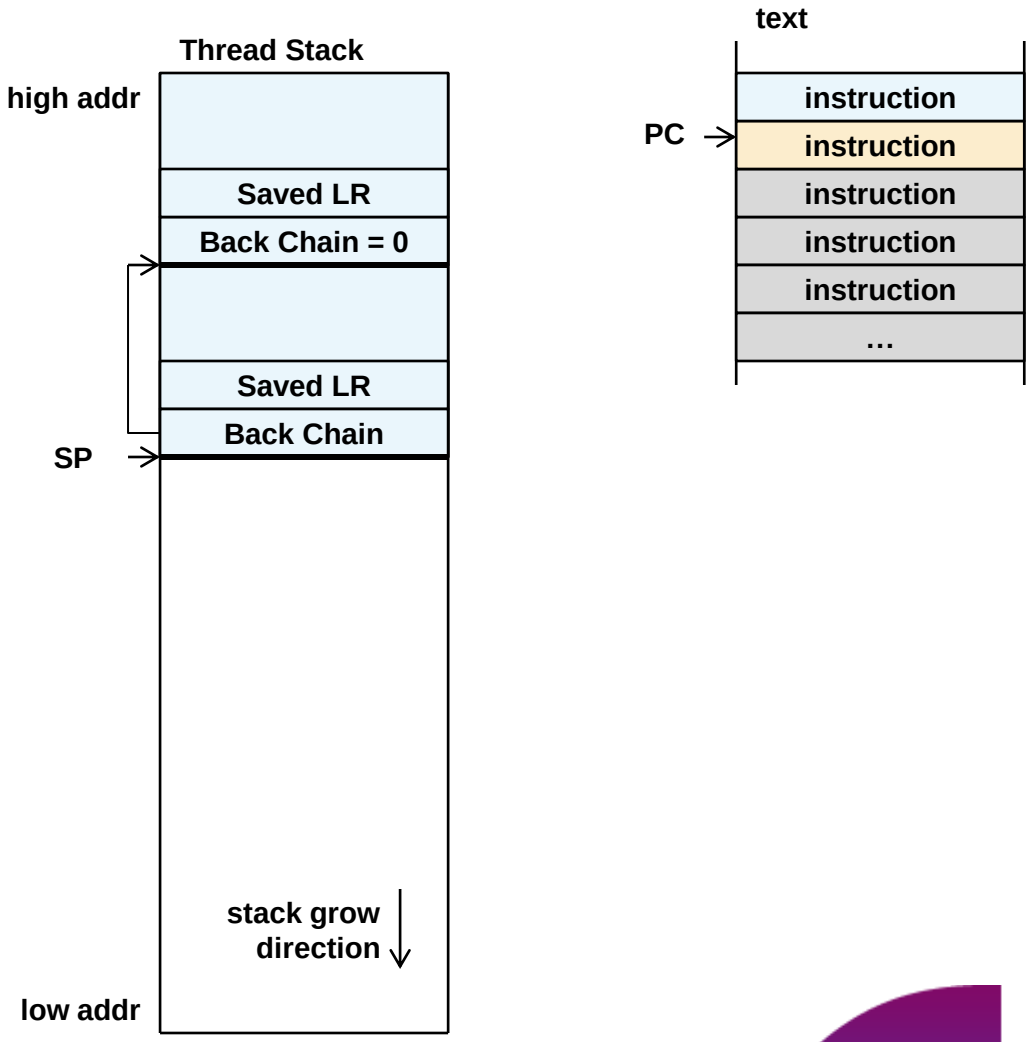
Request	Description
PEEKUSR, POKEUSR	Read/Write process data.
PEEKDATA , POKEDATA	Read/Write process memory.
ATTACH/DETACH	Attach to process, making the calling process the trace parent. This call will send a SIGSTOP to the process.
PTRACE_SINGLESTEP	Continues process until exit from system call, or after execution of single instruction.
PTRACE_CONT	Restarts stopped child process

ptrace essentially allows to attach to a process and to peek and poke at memory

Dynamic Tracing technique on Linux with ptrace
→ Inter-process Procedure Call using ptrace (used in all 3 stages)

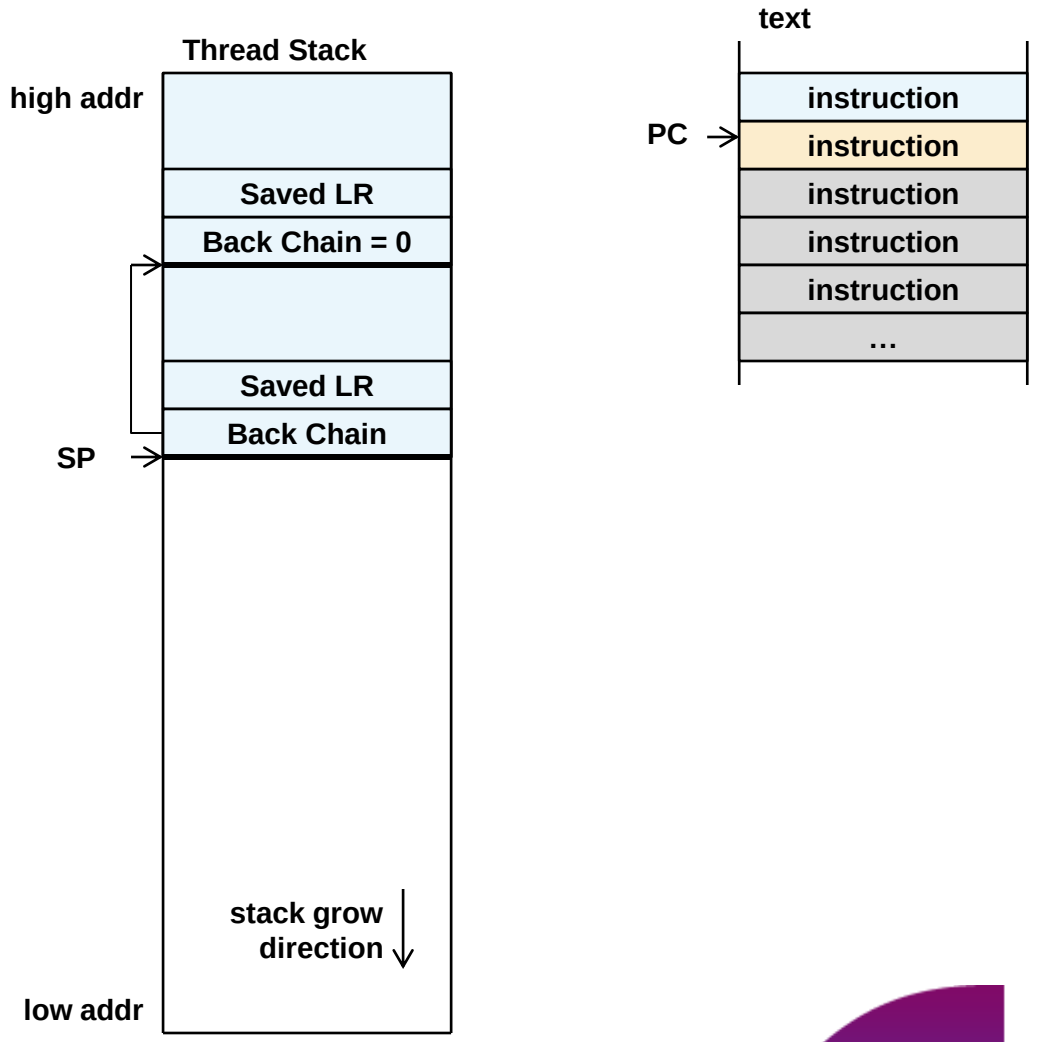
Process to trace is running

- Text segment
- Program counter
- Thread stack
- Stack Pointer



Dynamic Tracing technique on Linux with ptrace
→ Inter-process Procedure Call using ptrace (used in all 3 stages)

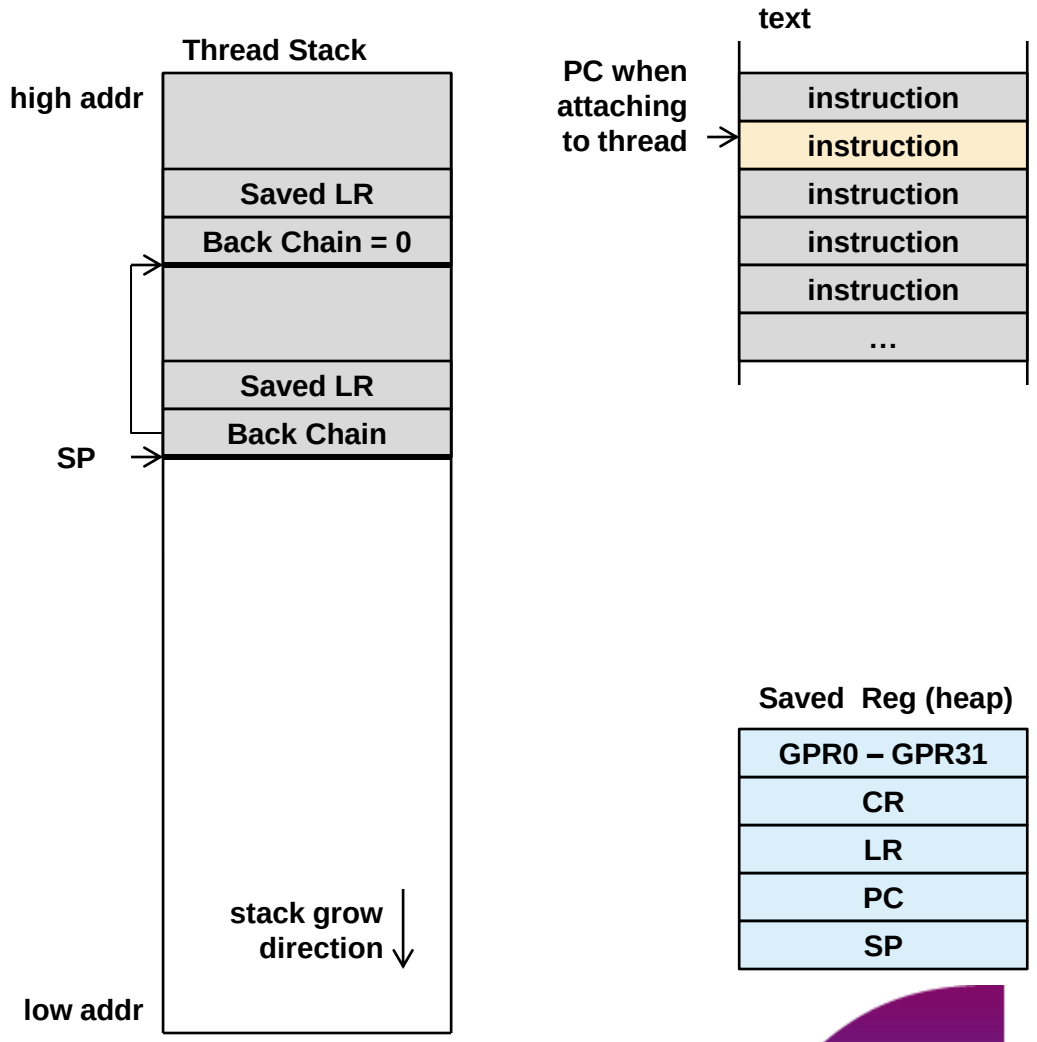
Sequence of steps to suspend a thread and make an inter-process procedure call:



Dynamic Tracing technique on Linux with ptrace
→ Inter-process Procedure Call using ptrace (used in all 3 stages)

Sequence of steps to suspend a thread and make an inter-process procedure call:

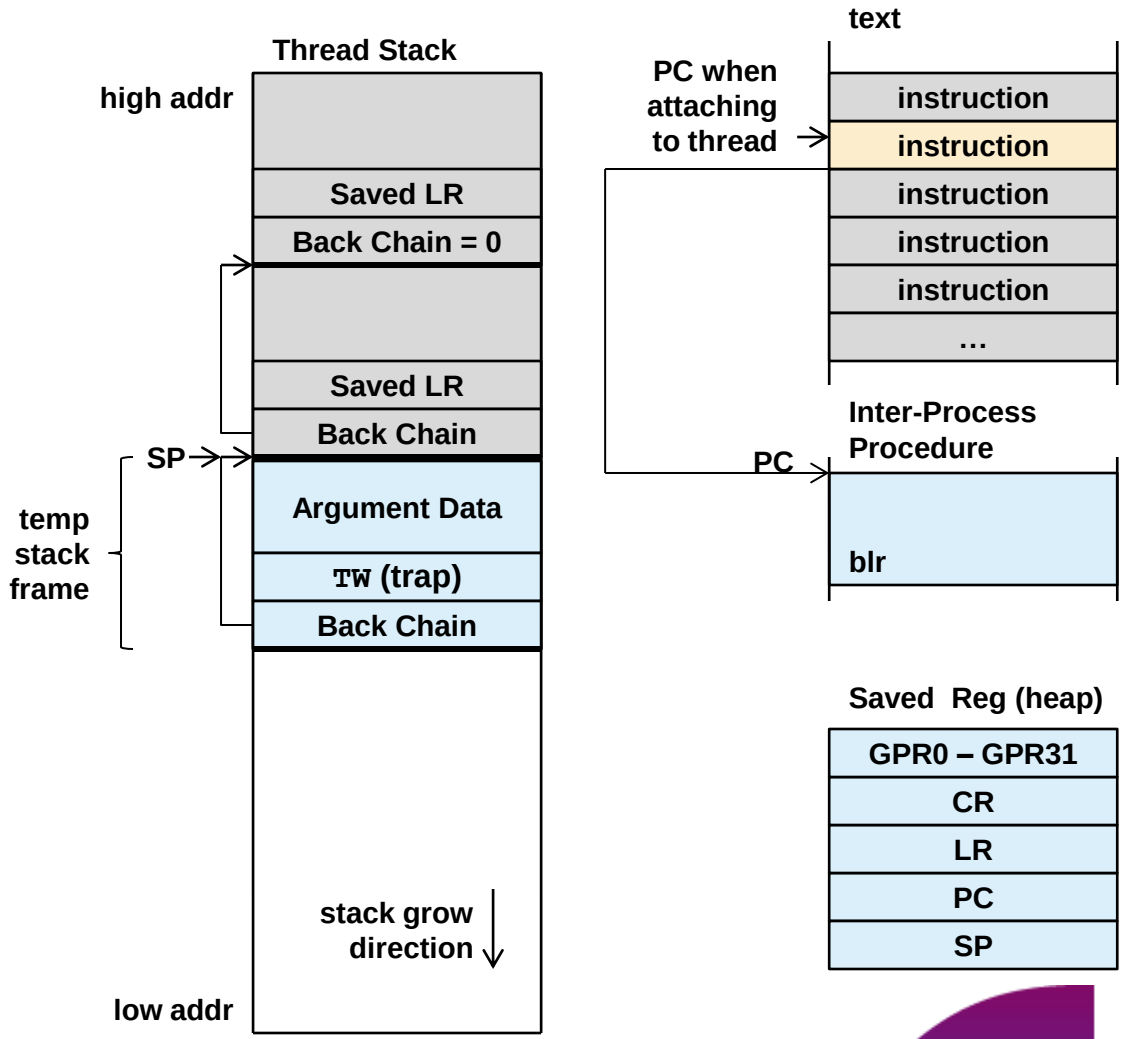
- Attach to thread (ptrace → ATTACH)
- Fetch register content (ptrace)
- Save their values to heap for future use
  - Including PC of currently running thread



Dynamic Tracing technique on Linux with ptrace
→ Inter-process Procedure Call using ptrace (used in all 3 stages)

Sequence of steps to suspend a thread and make an inter-process procedure call:

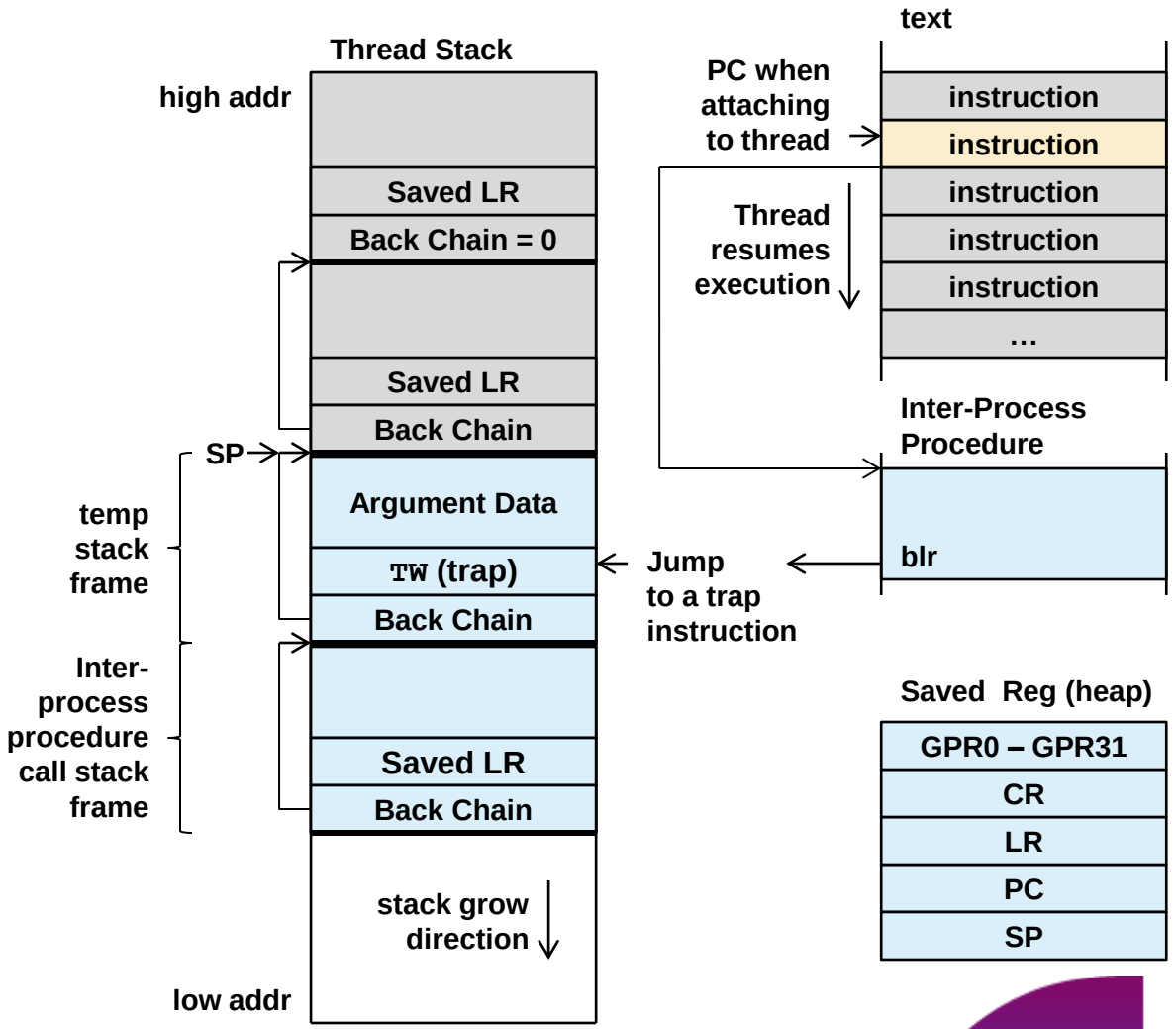
- Attach to thread (ptrace → ATTACH)
- Fetch register content (ptrace)
- Save their values to heap for future use
- Create a new temporary stack frame
  - A trap instruction is inserted in place of the “Saved LR”
- Setup processor to call the inter-process procedure with:
  - GPR3 to GPR8 are load with function parameters
  - argument data is loaded into the temp stack frame
  - PC is configured with address of function to call
  - thread execution is resumed (ptrace → PTRACE_CONT)



Dynamic Tracing technique on Linux with ptrace
→ Inter-process Procedure Call using ptrace (used in all 3 stages)

Sequence of steps to suspend a thread and make an inter-process procedure call:

- Attach to thread (ptrace → ATTACH)
- Fetch register content (ptrace)
- Save their values to heap for future use
- Create a new temporary stack frame
  - A trap instruction is inserted in place of the “Saved LR”
- Setup processor to call the inter-process procedure with:
- Inter-process call is now running
  - a new stack frame is created
- Function return (stack frame is popped from the stack)
- Branch to LR (Link Register) → this jumps to a trap instruction
- Trap Handling invokes ptrace
  - restores the CPU context with the previously saved registers
  - thread execution is then resumed from original PC location (ptrace → PTRACE_CONT)
- Thread execution is now resumed to its original location prior to inter-process procedure call



Dynamic Tracing technique on Linux with ptrace
→ Tracepoint Installation (2nd stage)

Now that we have our tracepoints, it's time to copy them over to the process memory.

For this we need space and for that we look at the processes memory map:

```
bash-4.1# cat /proc/3550/maps
00100000-00102000 r-xp 00000000 00:00 0          [vdso]
0f9c0000-0fb34000 r-xp 00000000 08:01 428263     /lib/libc-2.14.1.so
0fb34000-0fb43000 ---p 00174000 08:01 428263     /lib/libc-2.14.1.so
0fb43000-0fb45000 r--p 00173000 08:01 428263     /lib/libc-2.14.1.so
0fb45000-0fb48000 rwxp 00175000 08:01 428263     /lib/libc-2.14.1.so
0fb48000-0fb4b000 rwxp 00000000 00:00 0
0fb5b000-0fb74000 r-xp 00000000 08:01 428349     /lib/libgcc_s.so.1
...
0ff2e000-0ff2f000 r--p 0000a000 08:01 251006     /usr/lib/libmemTrace.so
0ff2f000-0ff30000 rwxp 0000b000 08:01 251006     /usr/lib/libmemTrace.so
0ff30000-0fff0000 rwxp 00000000 00:00 0
*** unreseved memory ***
10000000-10342000 r-xp 00000000 08:01 10386      bin/bbb-xc
10351000-103a9000 rwxp 00341000 08:01 10386      bin/bbb-xc
103a9000-105ab000 rwxp 00000000 00:00 0          [heap]
*** unreseved memory ***
48000000-4801f000 r-xp 00000000 08:01 428346     /lib/ld-2.14.1.so
```

Dynamic Tracing technique on Linux with ptrace
→ Enabling Instrumentation (3rd stage)

To this point we have not altered execution or impacted the performance of the target process.

Now we must insert the branch instructions at the start and end of every function to their matching tracepoint.

- Suspend all threads in process by attaching via ptrace.

How to get the list of threads for a specific process:

```
bash-4.1# ls /proc/3550/task
3550  3765  3767  3769  3771  3774  3776  3778  3780  3782  4026  4028  4030
3567  3766  3768  3770  3772  3775  3777  3779  3781  3783  4027  4029  6910
```

Dynamic Tracing technique used on Linux with LTTng

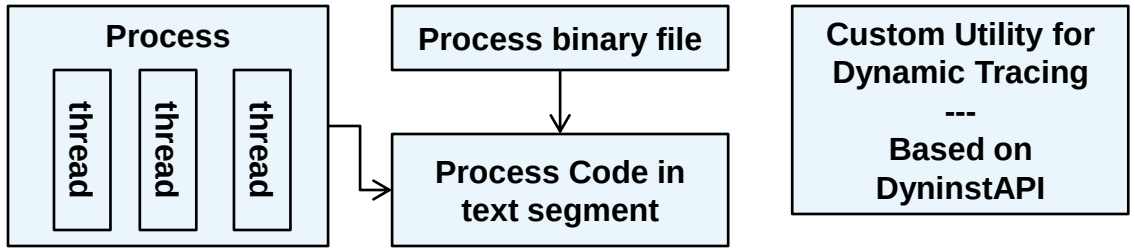
2nd approach based on the DyninstAPI library

Dynamic Tracing technique on Linux with Dyninst

Dyninst API is a dynamic instrumentation library developed by the Paradyn project

- Executable binary file
- Process running the program
- Program loaded in RAM

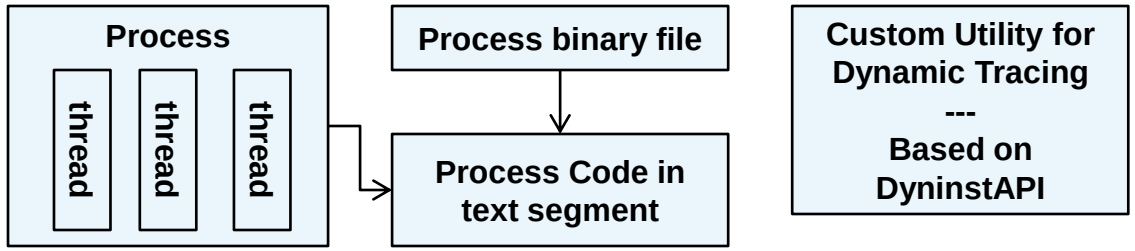
- Tracing utility

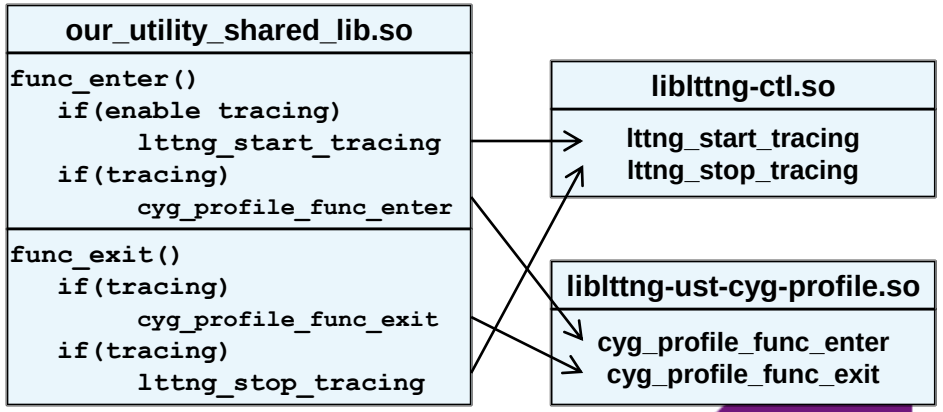


Dynamic Tracing technique on Linux with Dyninst

Sequence of steps to enable tracing using DyninstAPI

1. **Load shared libraries**
   - liblttng-ust-cyg-profile.so
     - cyg_profile_func_enter
     - cyg_profile_func_exit
   - liblttng-ctl.so
     - lttng_start_tracing
     - lttng_stop_tracing
   - our_utility_shared_lib.so
     - Entry function to invoke cyg_profile_func_enter
     - Exit function to invoke cyg_profile_func_exit
     - Additional control is added to enable / disable tracing on demand



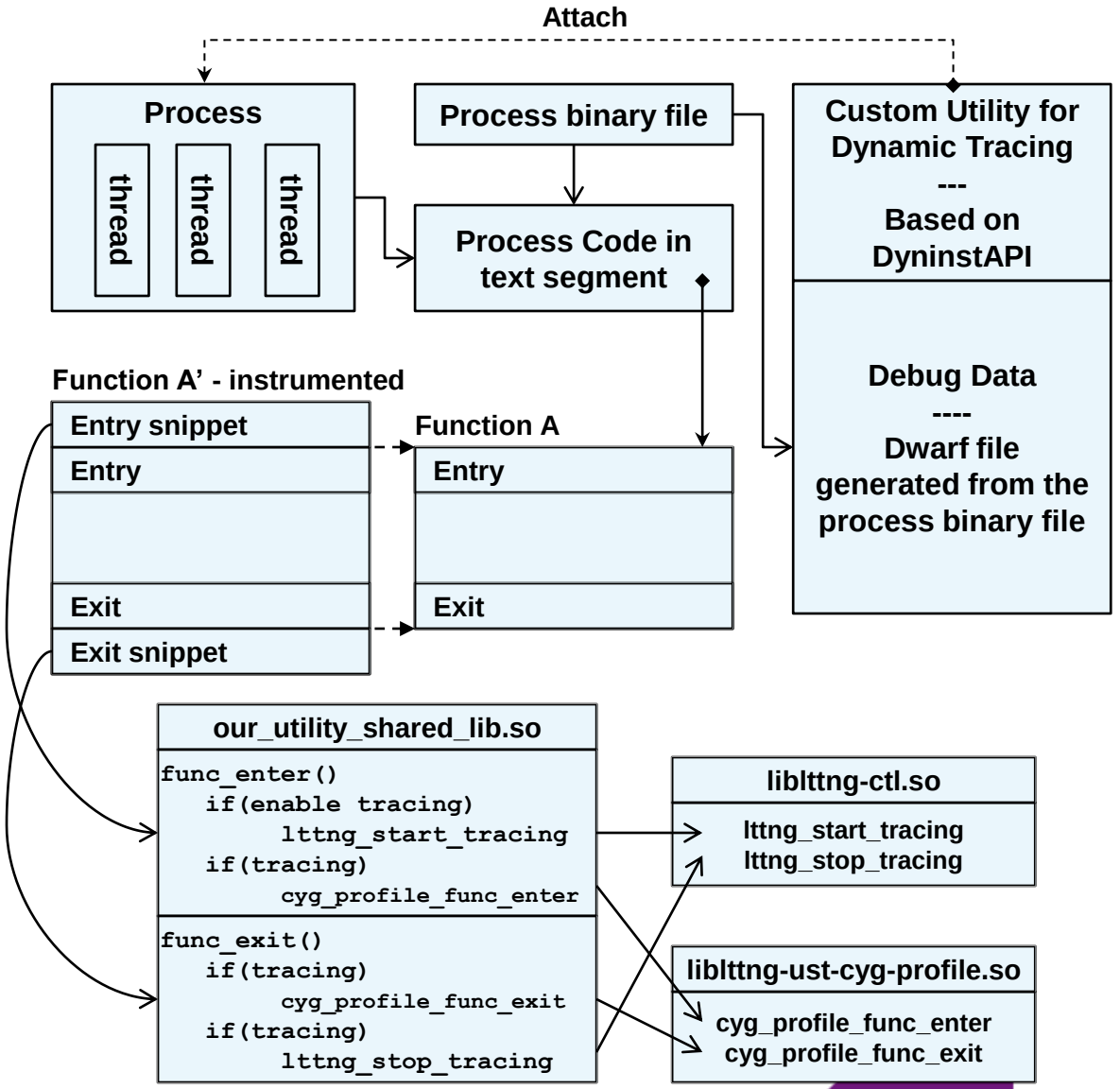


Dynamic Tracing technique on Linux with Dyninst

Sequence of steps to enable tracing using DyninstAPI

1. **Load shared libraries**
2. **Attach to the process**
3. **Find functions in our_utility_shared_lib.so**
4. **Find functions in Process binary file**
   - Generated debug data → dwarf file
5. **Begin insertion set**
6. **Create Dyninst insertion points and insert entry/exit snippets**
   - `this_fn` and `call_site` are needed for each function
7. **Finalize insertion set**

   Function A is replaced by Function A'
8. **Detach → process resumes execution**



Dynamic Tracing technique used on VxWorks

Dynamic Tracing technique used on VxWorks

Instrumentation is inserted by latching on function calls (as opposed to function entry and exit)

Function calls on 32-bit PowerPC

- Relative Calls:

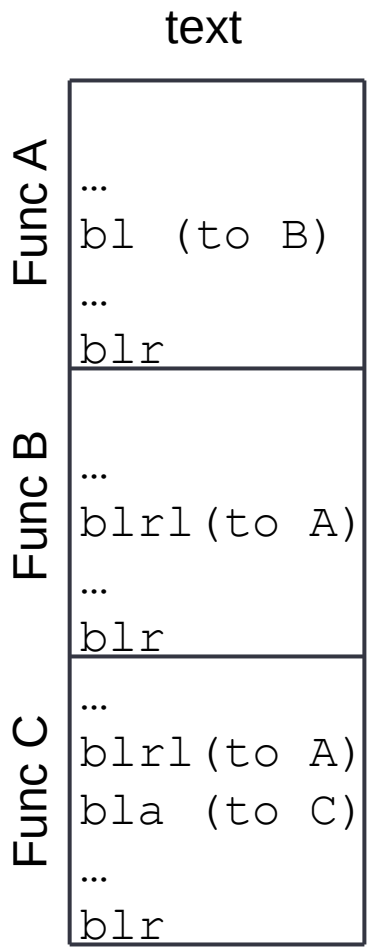
  LI: signed 24-bit value concatenated w/ 0b00

  [Effectively ± 25 bits (± 32 MB)]

  - **`bl`** → Branch update LR (Link Register)
    - Relative to address of **`bl`** instruction
  - **`bla`** → Branch update LR (Link Register)
    - Relative to address 0x0000_0000
- Long Calls:
  - **`blrl`** → Branch to LR and update LR
    - The 32-bit LR contains the address of the callee and then LR is updated with the address of the instruction which follow the **`blrl`**

Return from function calls

- **`blr`** → Branch to LR (Link Register)
  - The 32-bit LR contains the return address



Dynamic Tracing technique used on VxWorks

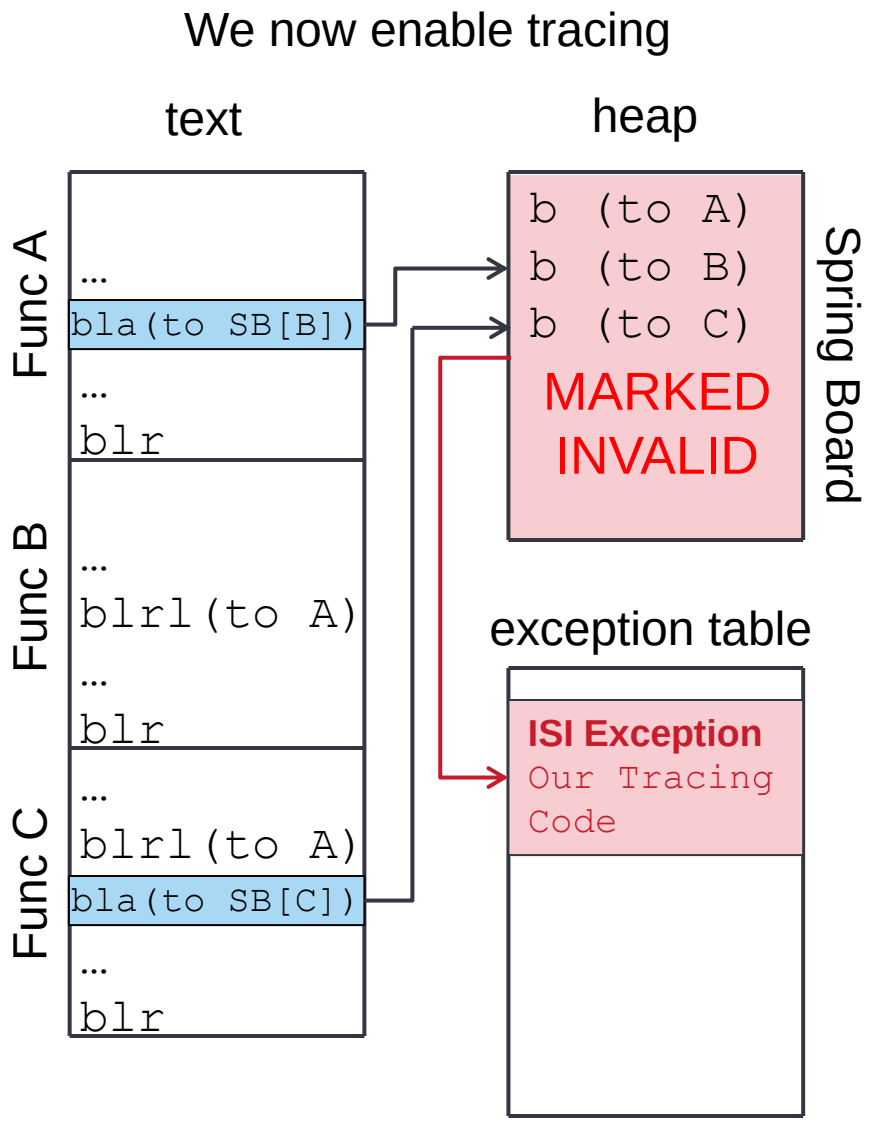
Our goal is to seamlessly trace all `bl`, `bla`, `blrl`, and `blr`

Technique used for `bl` and `bla` instructions

Our technique involves a Spring Board (SB), exception handling and tracing code

The Spring Board

- Used for replacing relative branches (**`bl`, `bla`**)
- Each callee needs its own entry in the SB
- An entry simple branch to the callee (without affecting the LR)
- When tracing is disabled:
  - We branch to the SB then branch to the callee
- When tracing is enabled:
  - The spring board is marked as invalid [through changing Block Address Translation (BAT) registers]
  - Now jumping to the SB causes an ISI Exception. The tracing code is in the exception table



Dynamic Tracing technique used on VxWorks

Technique used for `blrl` and `blr` instructions

The "spring board" approach has limitations for long branch instructions

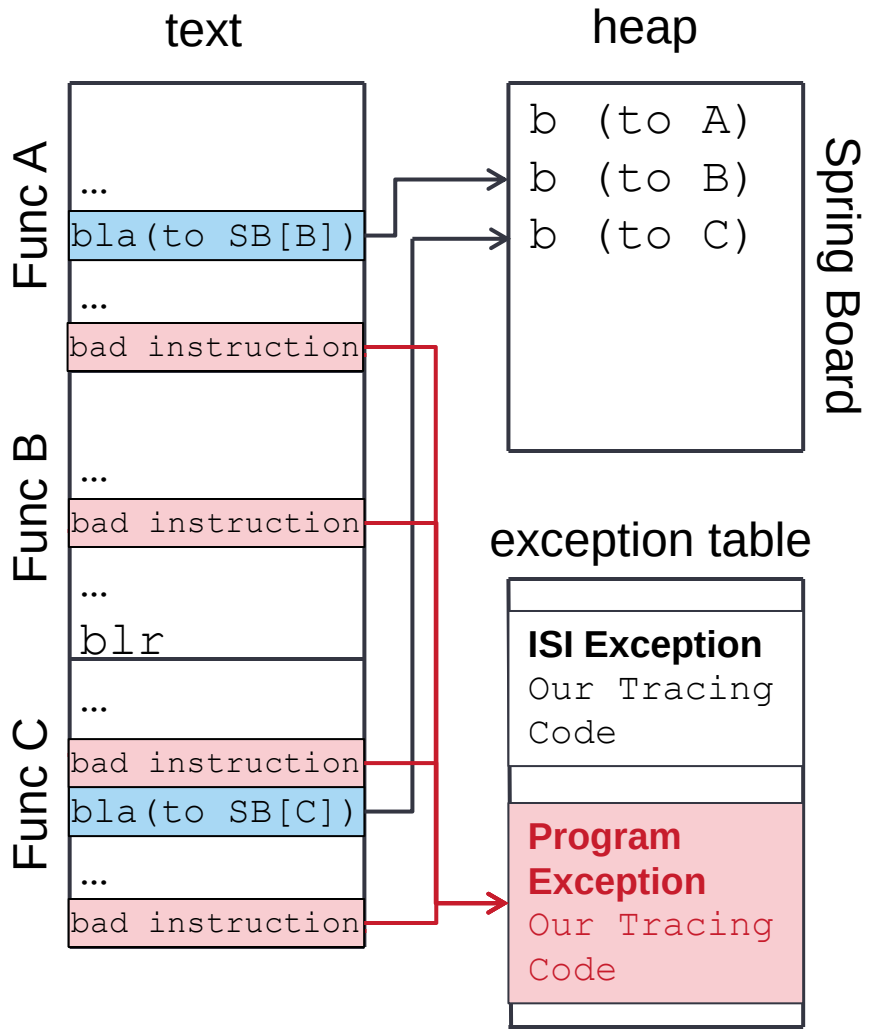
- A one-to-one SB entry would be required for each **`blrl`** and **`blr`** instructions
  - There is no practical way to know the value of LR in advance of code execution
  - Only one LR and we need to keep track of return address and address of callee
  - Space to allocate the SB is small

Our alternative is to replace all `blrl` and `blr` by an invalid instruction to generate an Program Exception

The drawback is that Program Exceptions are generated whether tracing is enabled or not

- But the exception handler is really simple (only a few instructions)

Enabling vs disabling of tracing is handled by the tracing code



Dynamic Instrumentation using GDB with LTTng

Dynamic Instrumentation using GDB with LTTng

For completeness we are mentioning this capability in our presentation

A quick google search yields the following results:

- https://suchakra.wordpress.com/2016/06/29/fast-tracing-with-gdb/
- http://www.slideshare.net/marckhouzam/gcc-summit2010-tmf
- http://git.lttng.org/?p=lttv.git;a=blob_plain;f=doc/developer/ust.html

Our next step would involve getting familiar with this new capability

Comparative Slides

Comparative Slides - Static Tracing Technique used on Linux with LTTng

PROS

- **Readily available and Open Source**
- **Portable to all architectures**
- **-finstrument-functions option is available since at least GCC 3.0.4**
- **LD_PRELOAD is a well-established ld-linux environment variable**

CONS

- **Requires recompilation**
- **Must traces all threads within a process**
- **Tracing is always on**

  doesn't provide controls to trigger start/stop of tracing

Comparative Slides - Dynamic Tracing technique on Linux with ptrace

PROS

- **Designed for low memory consumption and performance is considered**
- **Provides control to trace specific threads and to trigger start/stop of tracing**

CONS

- **Not easily portable to other architectures and challenging to implement**
- **Lots of caveats**

Comparative Slides - Dynamic Tracing technique on Linux with Dyninst

PROS

- **Dyninst is Open Source**
- **Dyninst is well documented and manuals are easily accessible from http://www.paradyn.org**
- **Our Dyninst program is simple and short: one file with about 200 lines of code**
- **Performance is taken into consideration**
- **Provides control to trace specific threads and to trigger start/stop of tracing**
- **Portable to every architecture supported by Dyninst**
- **Actively maintained**

CONS

- **Dyninst libraries are large (~100MB / disk space)**
- **Consumes a lot of memory**

Based on our experiment, instrumenting a 50 MB binary translated to 1.3 GB being used by our utility after calling the findFunction API

Comparative Slides - Dynamic Tracing technique used on VxWorks

PROS

- **Designed for low memory consumption and performance**

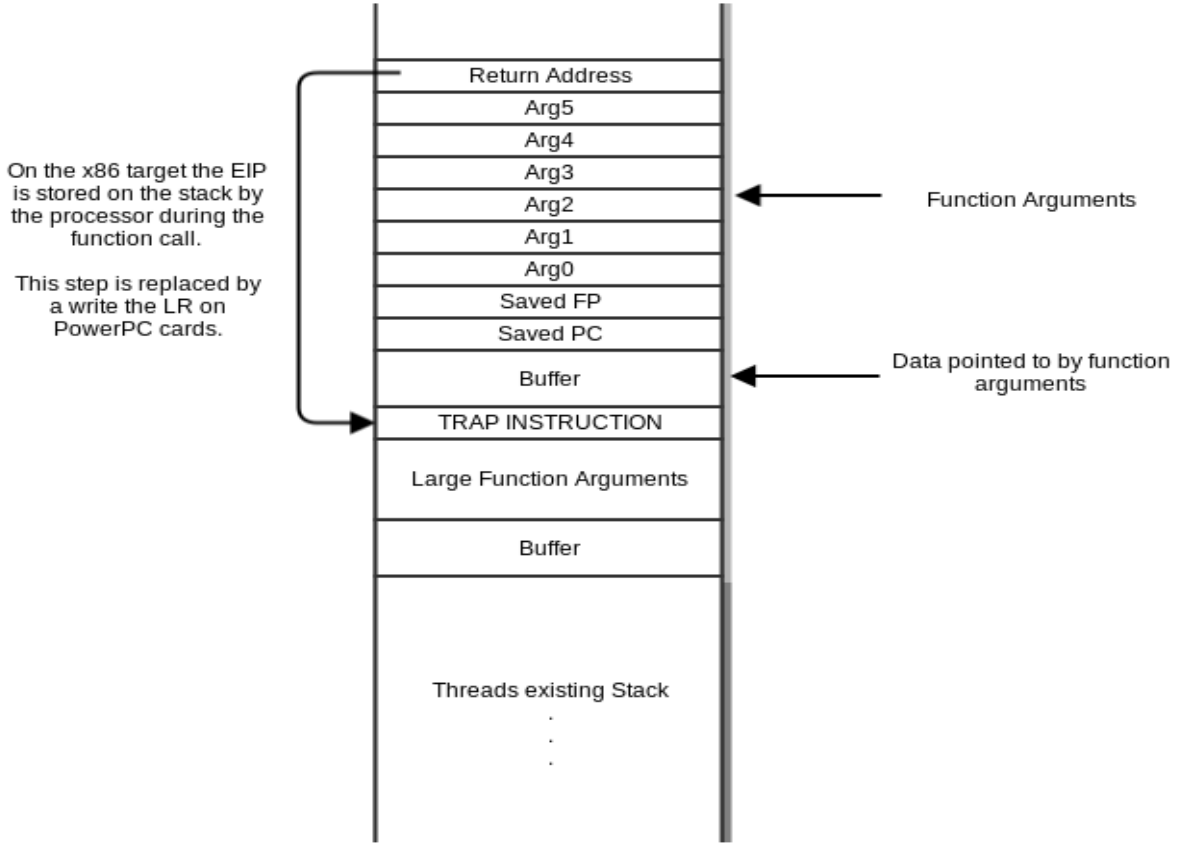
CONS

- **Approach very targeted to VxWorks - not proven in Linux land**
- **Not easily portable to other architectures and challenging to implement**

Thank You

Backup slides

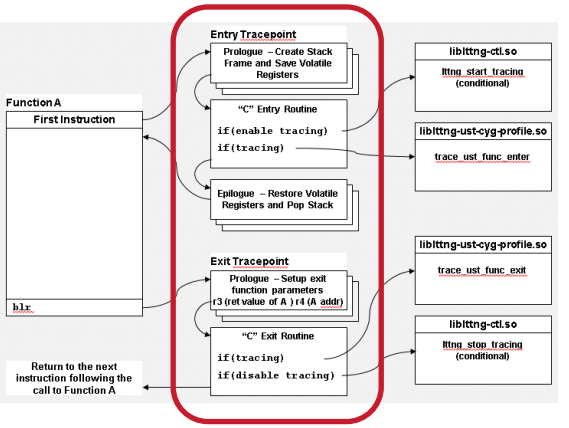
Dynamic Tracing technique on Linux with ptrace
→ Inter-process Procedure Call using ptrace (used in all 3 stages) x86



After suspending a thread, we:

- **Push argument data on the stack**
- **Push a trap instruction**
- **Then create a stack frame as would the compiler:**
  - push PC
  - push FP
  - push arguments
- **Push the EIP normally done by the processor on processing the call instruction.**
  - Address pushed is that of trap instruction.

Dynamic Tracing technique on Linux with ptrace
→ Autonomously Produced Tracepoint (2nd stage) PowerPC



Entry Tracepoint Function

```
EntryTrcFuncPrologue:

stwu        %r1, -80(%r1)
stw         %r0,   8(%r1)
mflr        %r0
stw         %r0,  12(%r1)
mfcr        %r0
stw         %r0,  16(%r1)
mfxer       %r0
stw         %r0,  20(%r1)
stw         %r0,  24(%r1)
stw         %r2,  28(%r1)
stw         %r3,  32(%r1)
...
stw         %r14, 76(%r1)
bl          4
mflr        %r0
stw         %r0,   4(%r1)

lis         %r3,  a.h
ori         %r3, %r3, a.l
lis         %r4,  x.h
ori         %r4, %r4, x.l
lis         %r5,  y.h
ori         %r5, %r5, y.l
mtlr        %r5
blrl    (*)
```

```
EntryTrcFuncEpilogue:

lwz         %r14, 76(%r1)
...
lwz         %r0,  24(%r1)
mtctr       %r0
lwz         %r0,  20(%r1)
mtxer       %r0
lwz         %r0,  16(%r1)
mtcr        %r0
lwz         %r0,  12(%r1)
mtlr        %r0
lwz         %r0,   8(%r1)
addi        %r1, %r1, 80
```

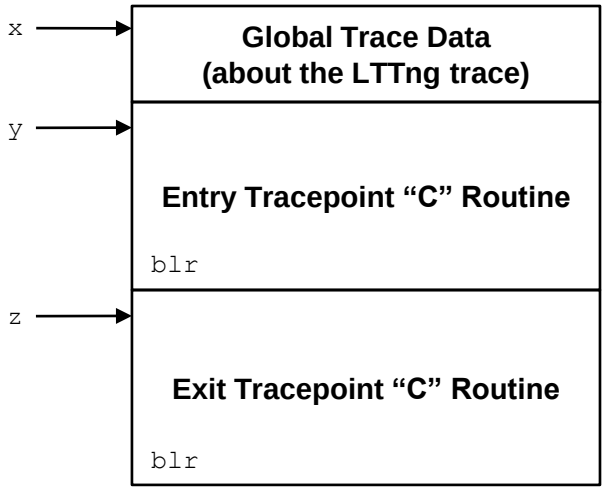
Instruction previously replaced by branch to wrapper

Branch back to next instruction in function

(*) next instruction is beginning of EntryTrcFuncEpilogue

Exit Tracepoint Function

```
ExitTrcFuncEpilogue:

lis         %r5,  a.h
ori         %r5, %r5, a.l
lis         %r6,  x.h
ori         %r6, %r6, x.l
lis         %r0,  z.h
ori         %r0, %r0, z.l
mtctr       %r0
bctr
```



a → address of function being traced
x → address of Global LTTng Trace Data
y → address of Entry Tracepoint "C" routine
z → address of Exit Tracepoint "C" routine

r3, r4, r5, r6
→ (registers used for parameter passing)

Entry Tracepoint "C" routine parameters
- r3 → a r4 → x

Exit Tracepoint "C" routine paramaters
- r3/r4 → traced function return value
- r5 → a r6 → x

Dynamic Tracing technique on Linux with ptrace
→ Autonomously Produced Tracepoint (2nd stage) x86