GDB as a versatile instrumentation server

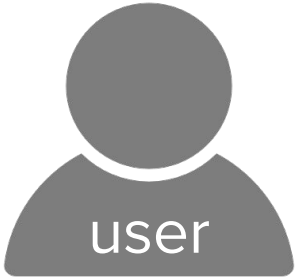
09-12-19

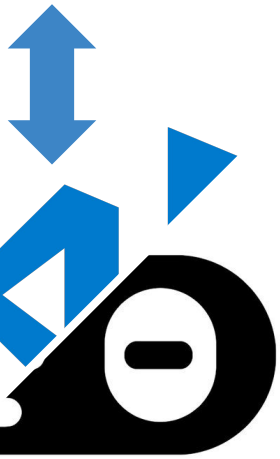
Paul NAERT
Pr Michel DAGENAIS

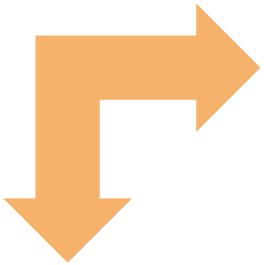
Summary

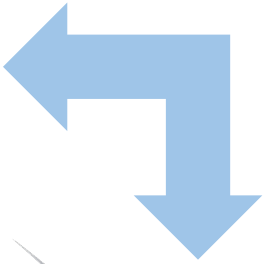
- Introduction
  - What do I mean by instrumentation server ?
  - Why use GDB for this purpose ?
  - Expanding a tool with GDB
- Dynamic instrumentation with GDB
- Use cases
  - Dynamic C/C++ tracing
  - Memory analysis
- Other work

GDB as a service



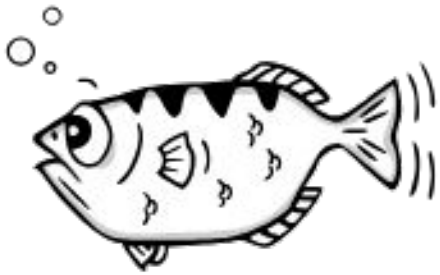






LSP

Editor



GDB

Language Server

Why go through GDB ?

- Client/Server architecture
- Attaching and detaching capabilities
- DWARF debug information integration
- Python interface
- Signal interception and handling
- Inferior function calling
- Dynamic library loading
- Compiling code in the inferior scope -> access to local variables

- *Dynamic instrumentation*

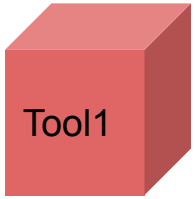
  Not upstream yet

Expanding the capabilities of current tools

Expanding a tool with GDB

Tool 1:

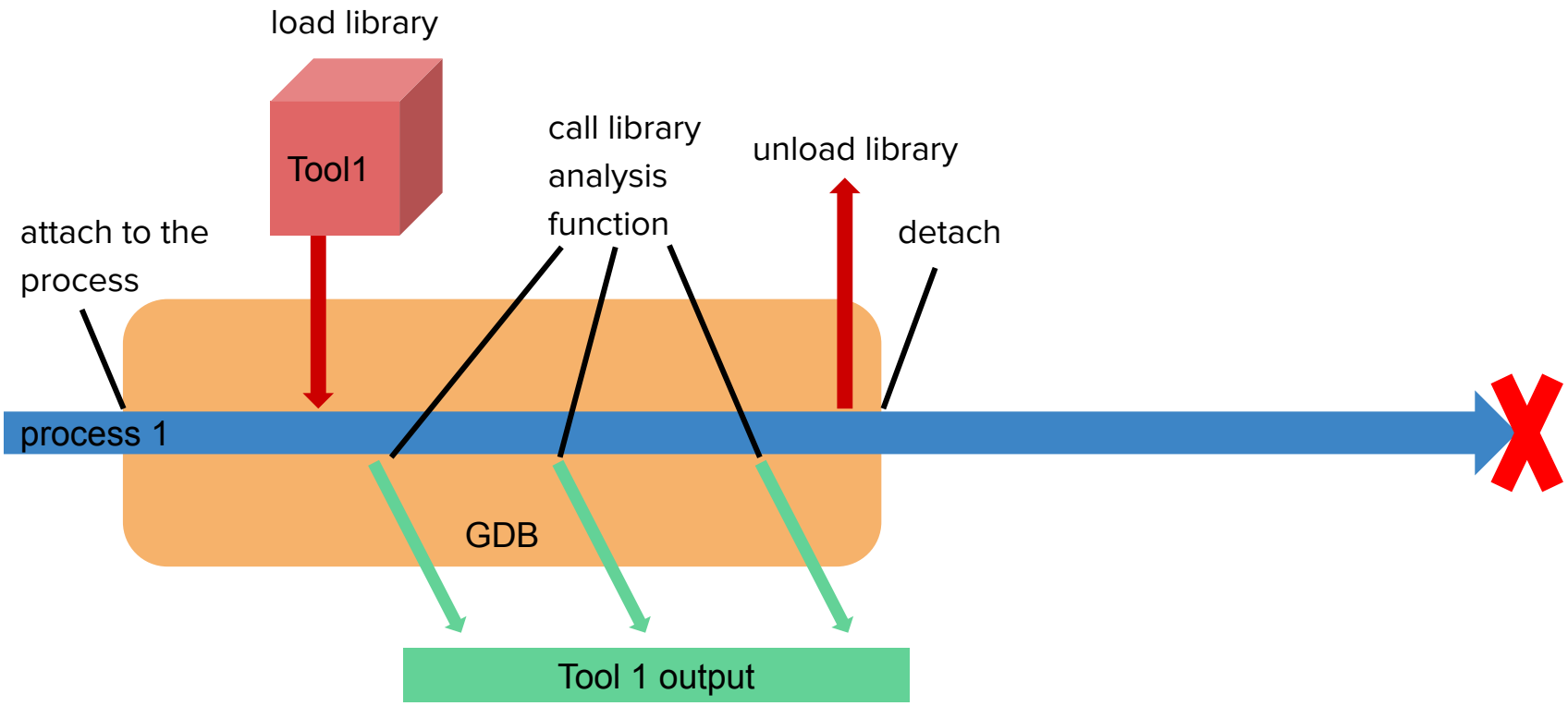
A memory checking library that monitors malloc and free and checks for memory leaks at the end of the program execution.



Process 1:

A long running process that sometimes crashes because it runs out of memory.

Expanding a tool with GDB



Expanding a tool with GDB

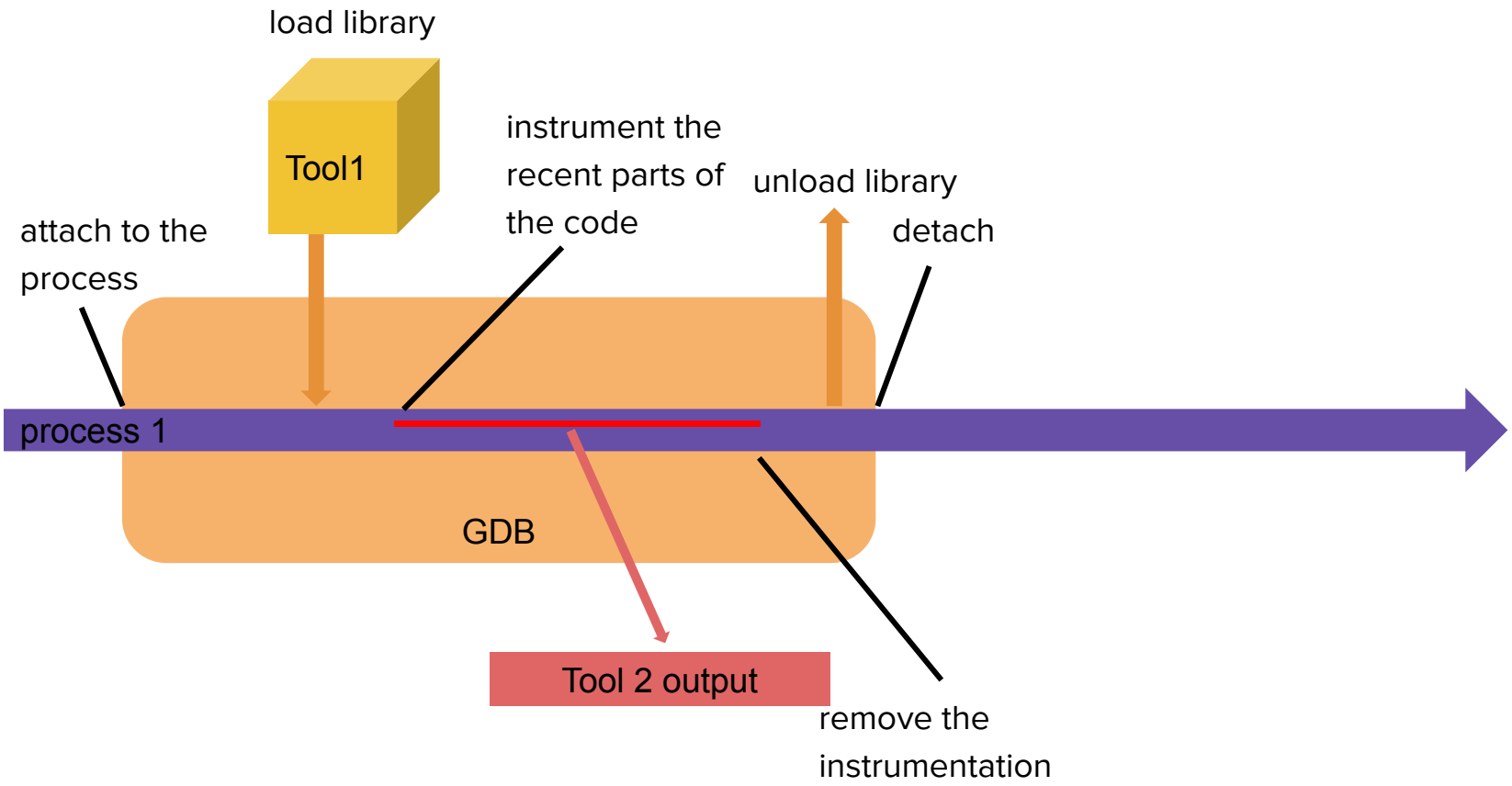
Tool 2:

A memory checking library that checks for memory corruption by instrumenting memory accesses.

Process 2:

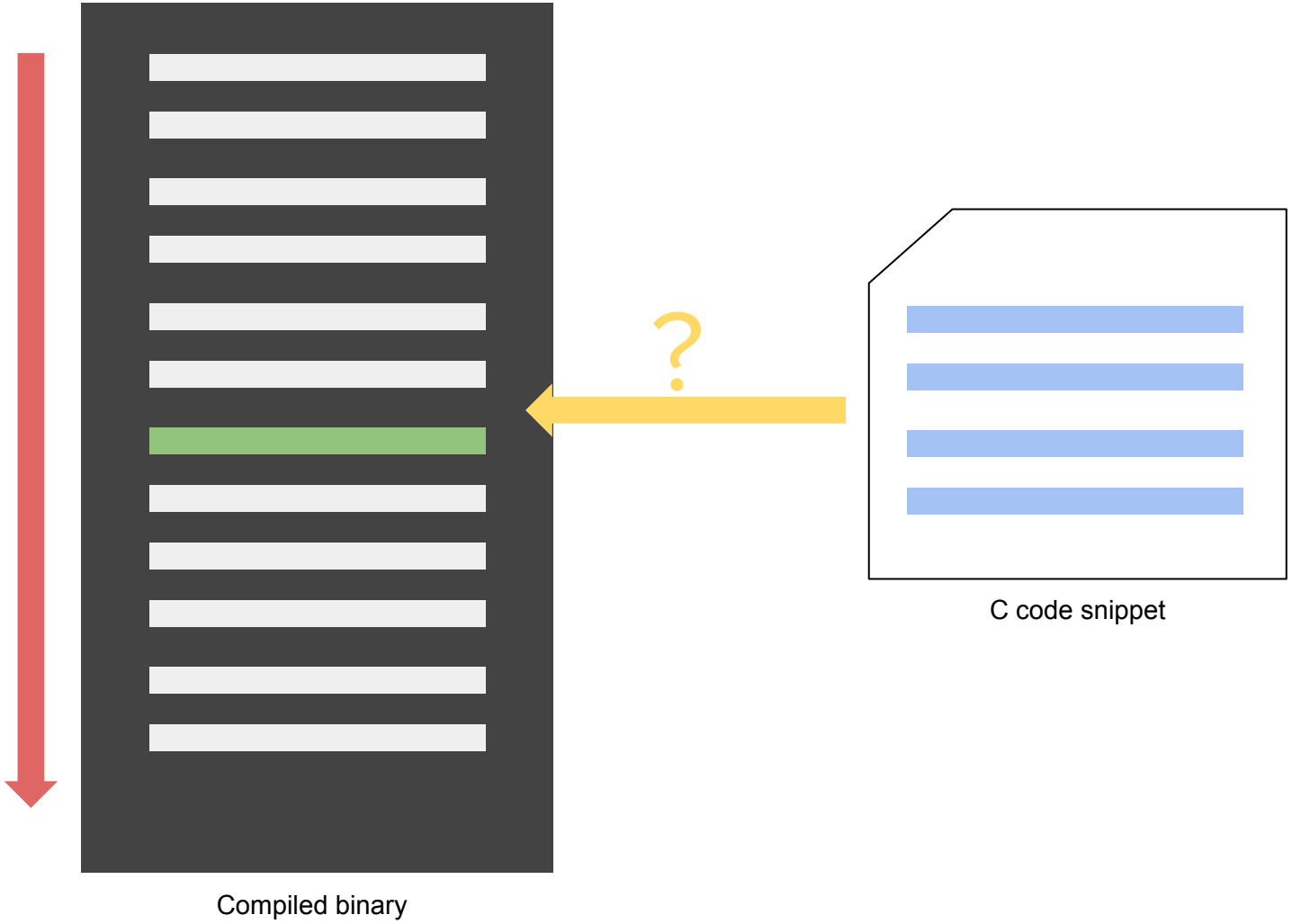
A long running process that produces invalid output due to memory corruption because of recent modifications.

Expanding a tool with GDB

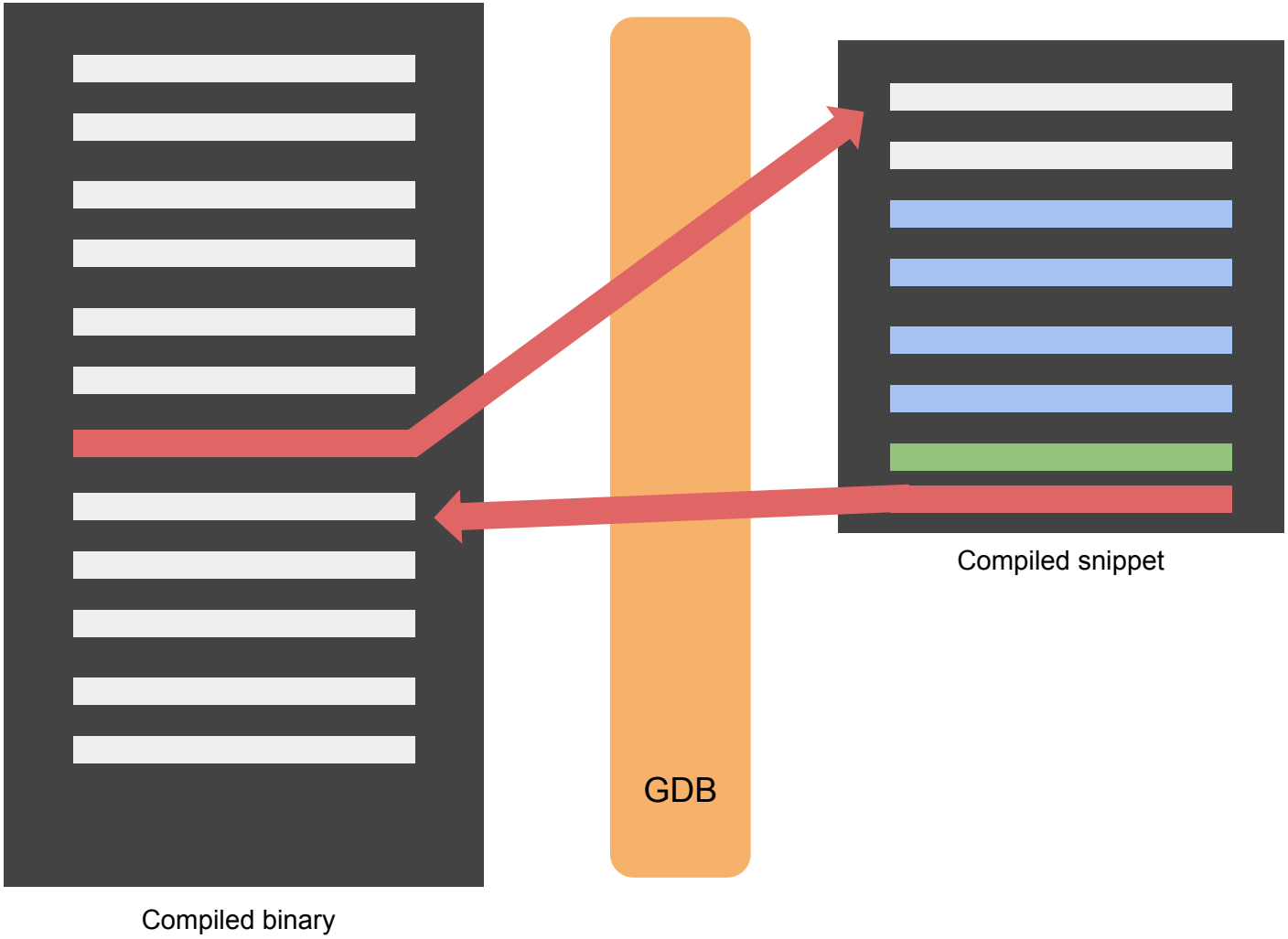


Dynamic instrumentation with GDB

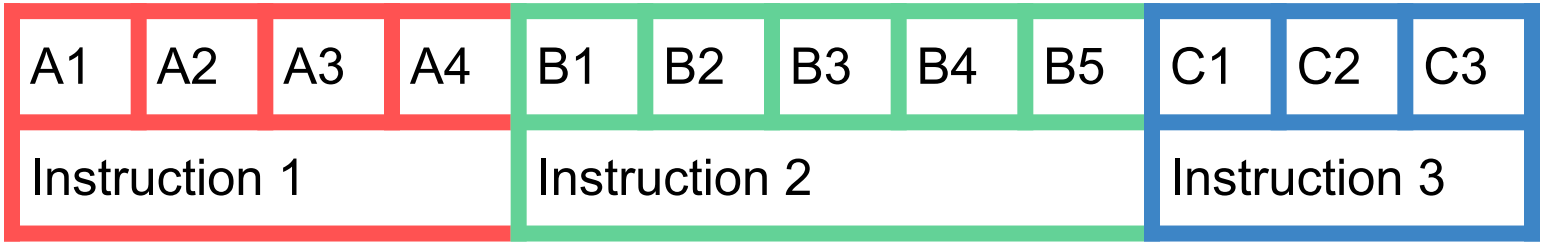
Dynamic code patching



Dynamic code patching

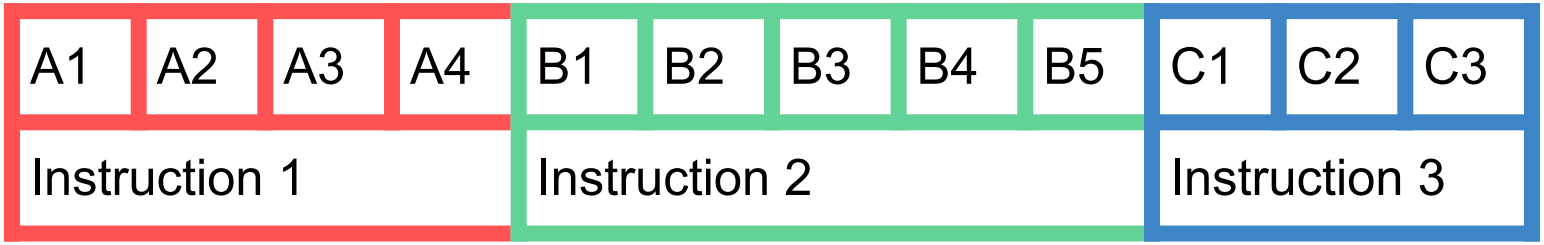


Dynamic code patching : x64 implementation



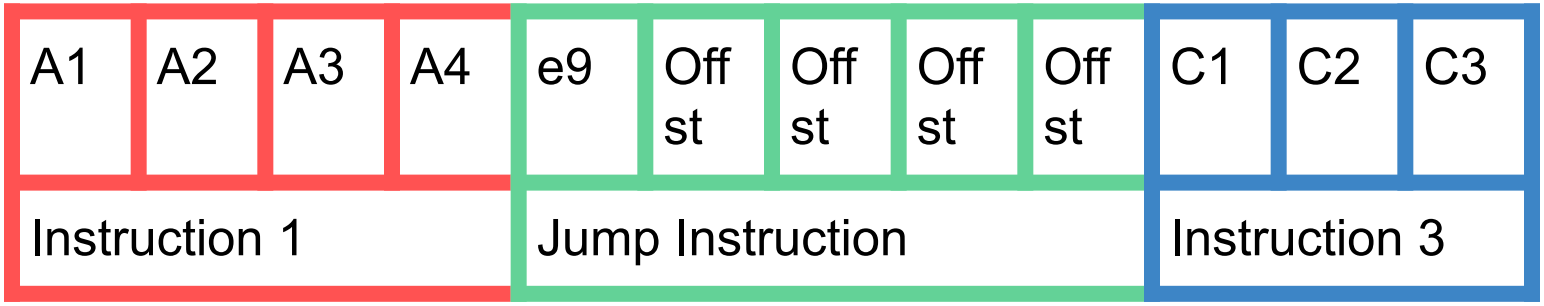
Patching code at instruction 2 :

Dynamic code patching : x64 implementation

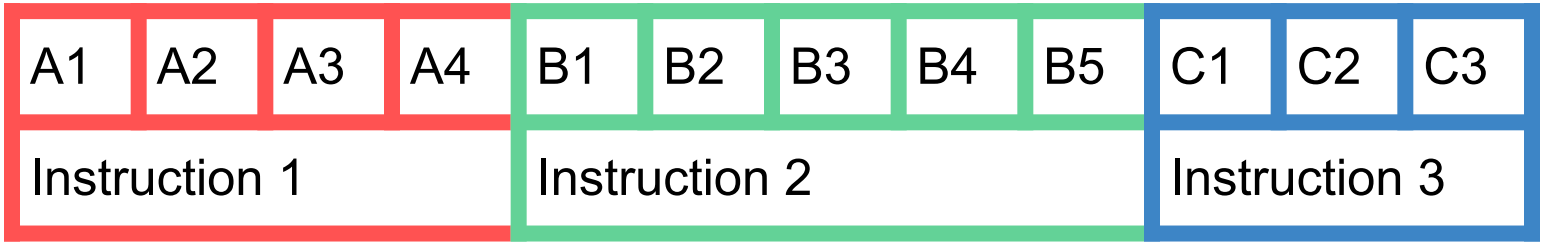


Patching code at instruction 2 :

Replace whole instruction with a jump :



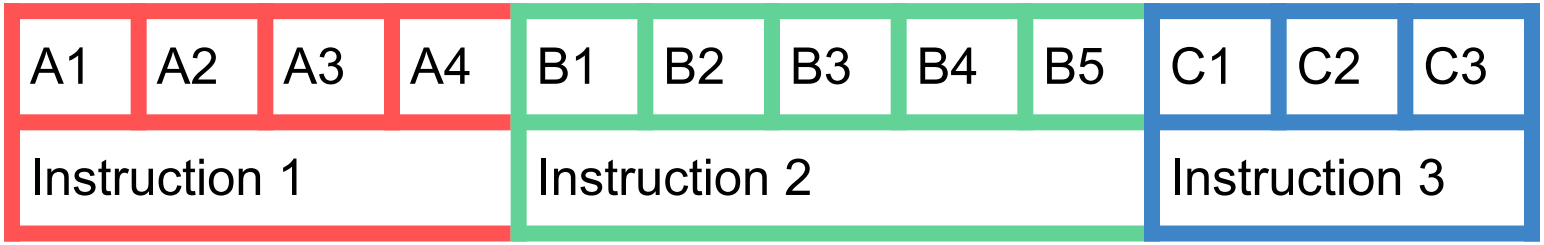
Dynamic code patching : x64 implementation



Patching code at instruction 1 :

Putting a 5 byte jump corrupts Instruction 2.

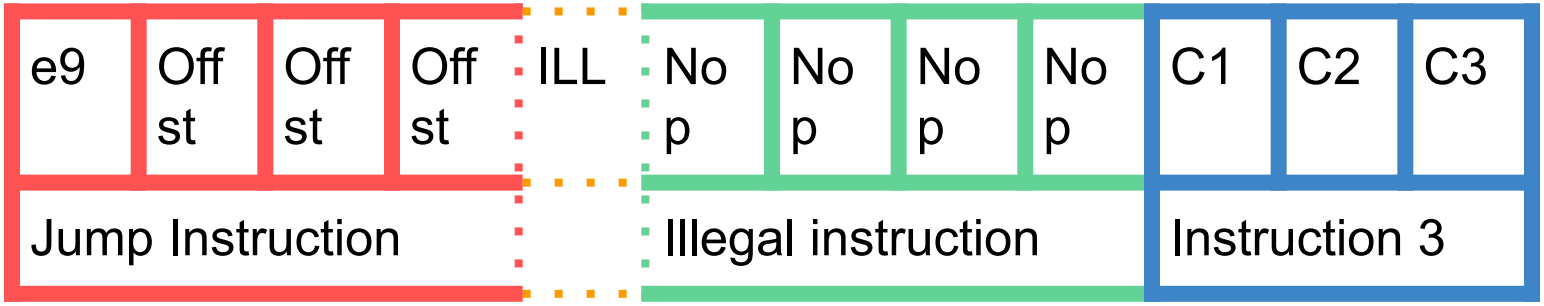
Dynamic code patching : x64 implementation



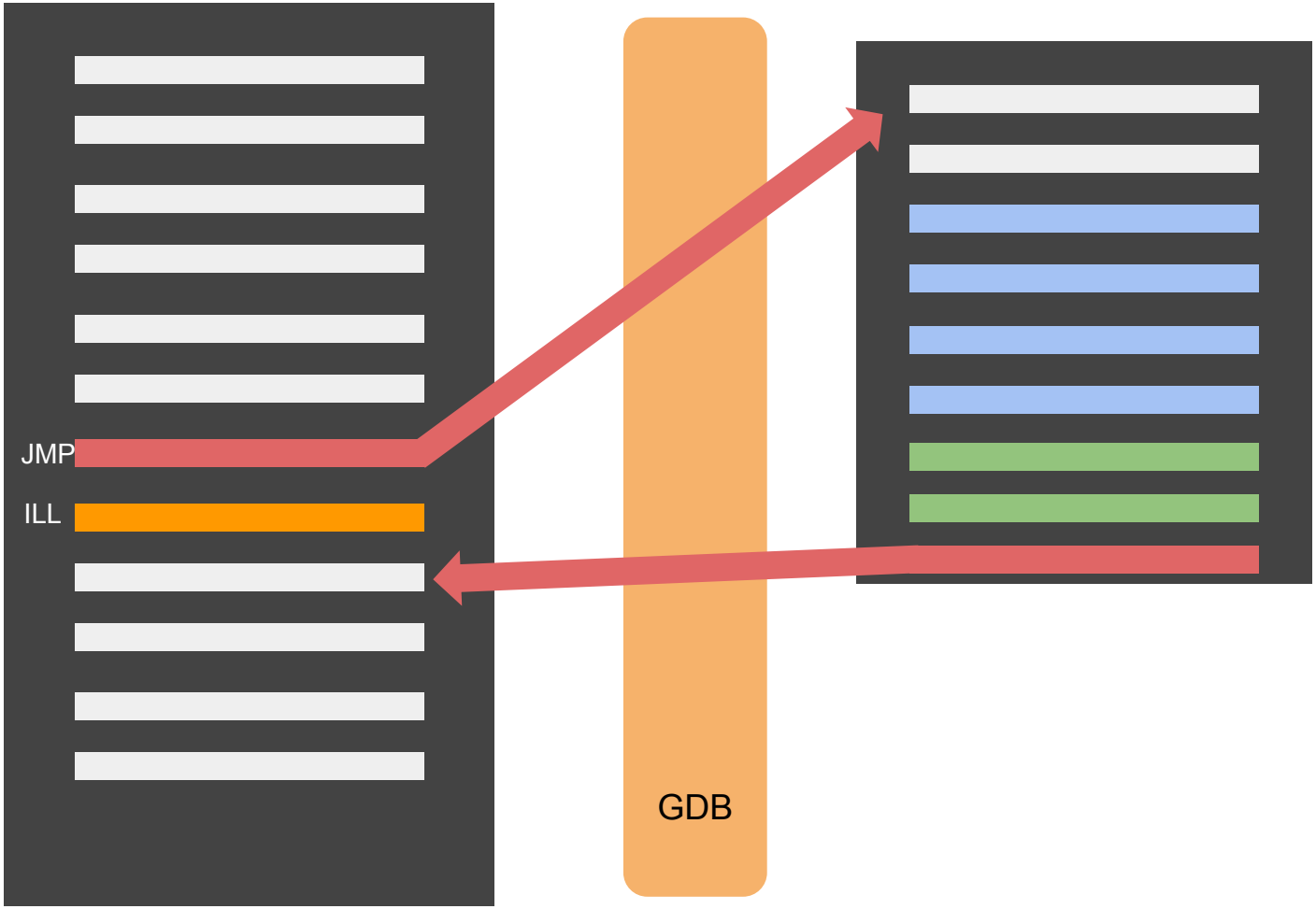
Patching code at instruction 1 :

Replace B1 with an illegal instruction, which is also part of the offset.

-> Need to be able to map pages at arbitrary locations

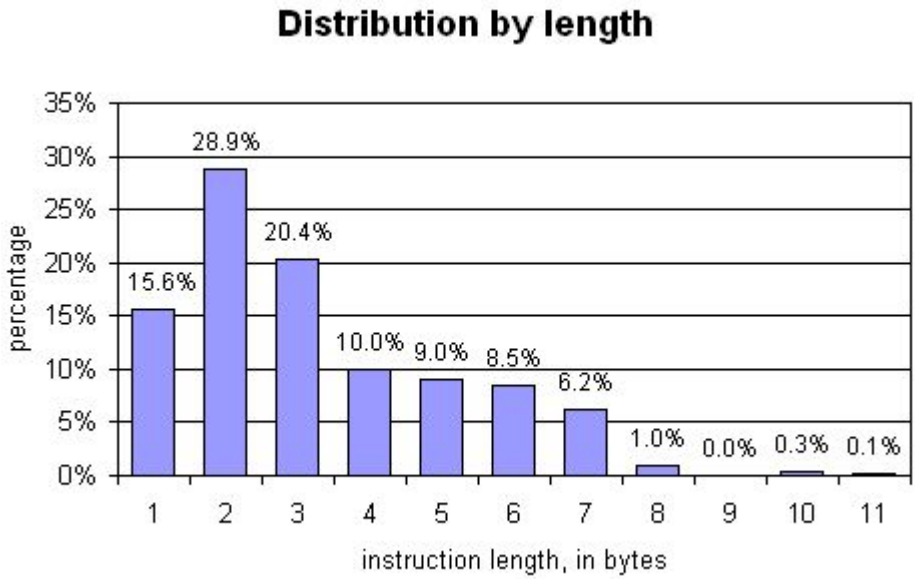


Patching short instructions



Patching short instructions

- around 60% of instructions are shorter than 5 bytes
- Virtually every address is now instrumentable



https://www.strchr.com/x86_machine_code_statistics

Instrumentation Performance on x64

- About 100 instructions overhead - 55ns on i7-4790 per instrumentation location

- Insertion time : 27ms per instrumentation location

- For reference :

  - getpid() system call : 350 - 1000ns per call.
  - breakpoint : 0.5 - 1ms to stop and resume the inferior

Examples of use cases

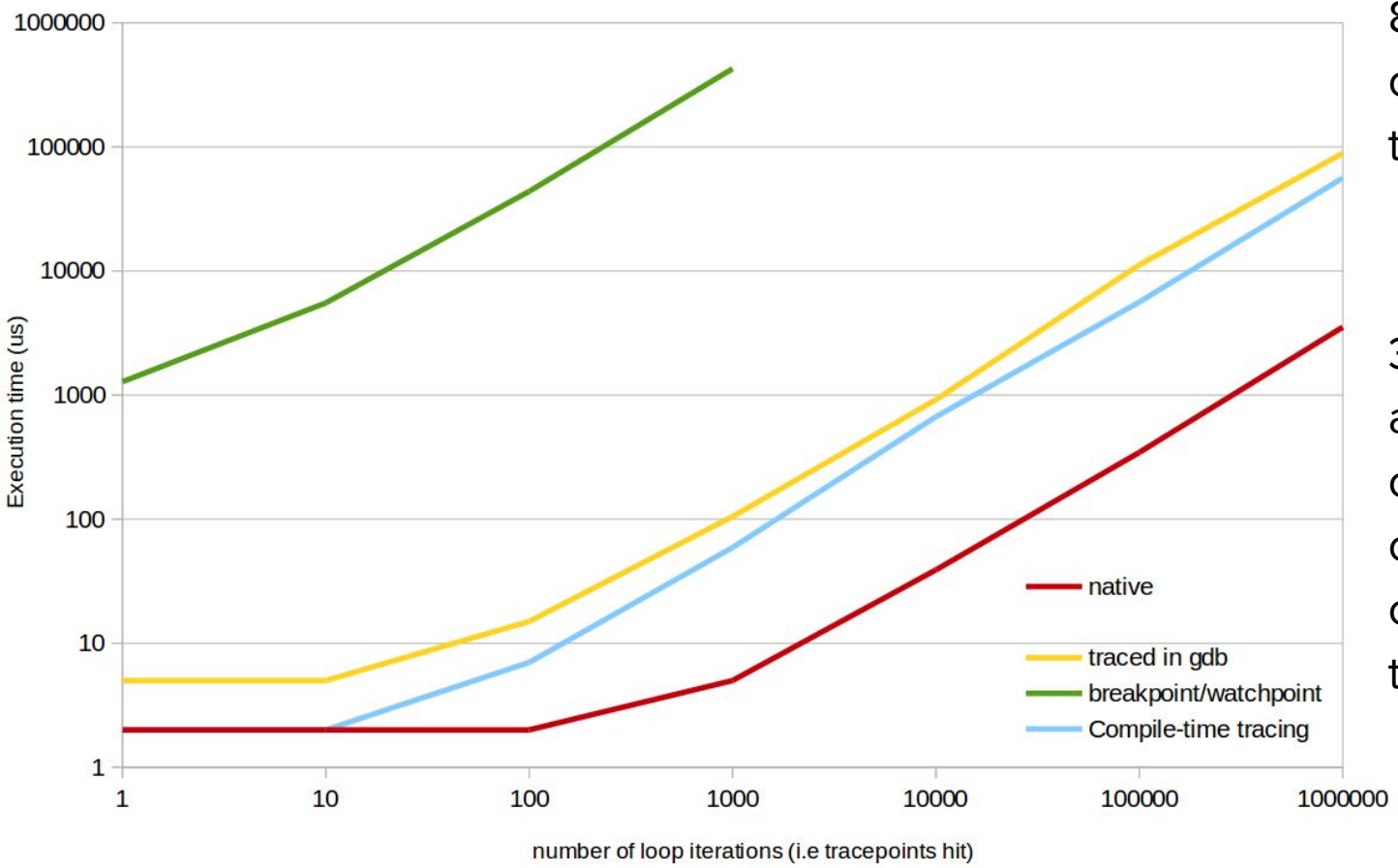
Dynamic C/C++ tracing

- Information about a process execution

- Existing solutions
  - Static (e.g. LTTng)
  - Slow (e.g. GDB breakpoints)
  - Limited (e.g. DynTrace)

Dynamic C/C++ tracing

Tracing overhead on a simple program

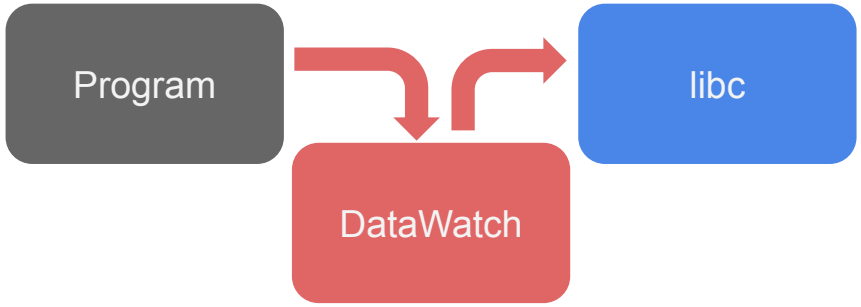


85ns average overhead per tracepoint

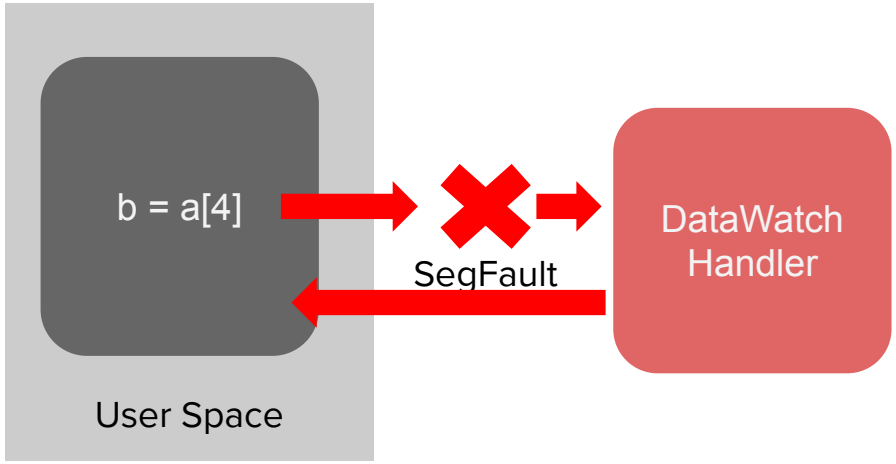
30-40ns average overhead compared to compile-time tracing

Memory analysis : a user-space only Data Watch

1. Override malloc() and free()
   malloc() now adds information in the most significant bits of the address, and stores what has been allocated and where.



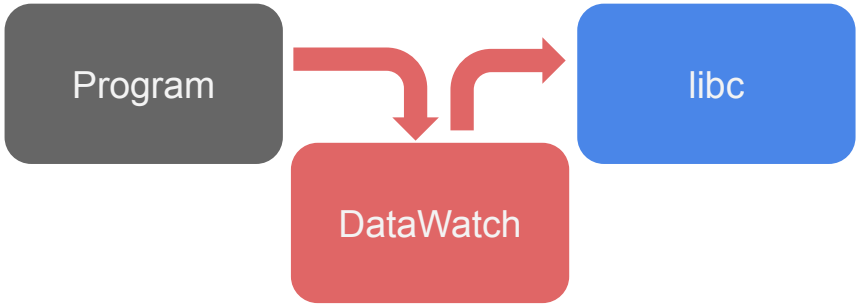
2. Each pointer resolution now raises a segmentation fault, which is handled by DataWatch. If the dereference is within bounds, it corrects the address and sends back the value.

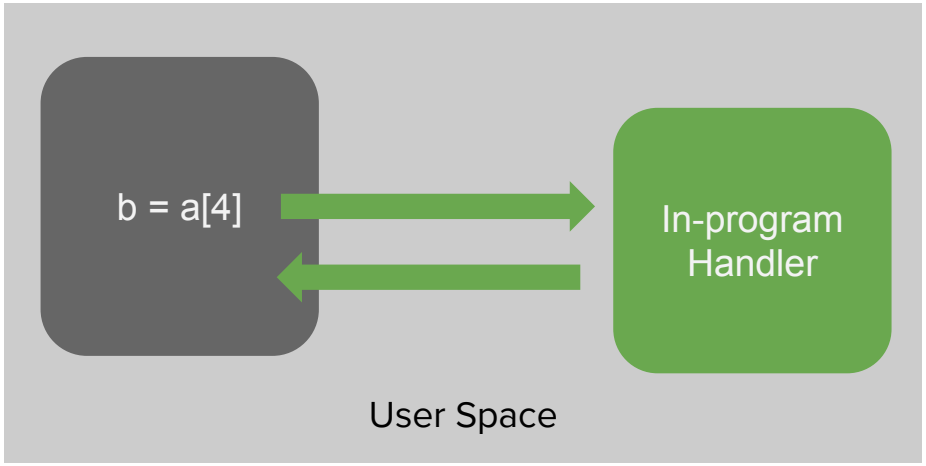


Memory analysis : a user-space only Data Watch

1. Override malloc() and free()
   malloc() now adds information in the most significant bits of the address, and stores what has been allocated and where.

2. The first memory access will cause a Segfault, and GDB patches that instruction so that subsequent calls will not generate a signal.
   The resolution is corrected in the program without any transition to kernel space.





Memory analysis : a user-space only Data Watch

Advantages :

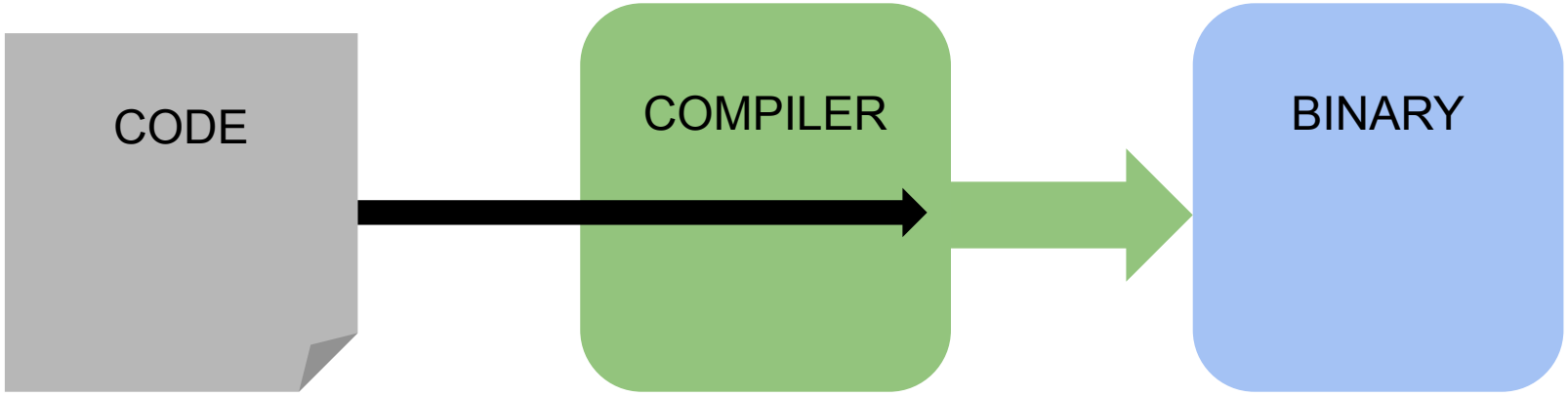
1. Can benefit from GDB's client/server architecture
2. It can target only a specific part of a program.
3. Can be attached to a running binary, although it will not check already allocated memory.
4. Can work in conjunction with Data Watch if pointers are shared outside of the targeted area.
   needs a Kernel module

Limitations :

1. Overhead can be significant in libc : giving an invalid address to strcpy will cause a large number of illegal instructions to be hit.
2. No complete override of malloc and free : memory allocated outside of the target range will not be checked.
3. No verification for data allocated on the stack.
4. Issues with system calls and ioctl without a kernel module.

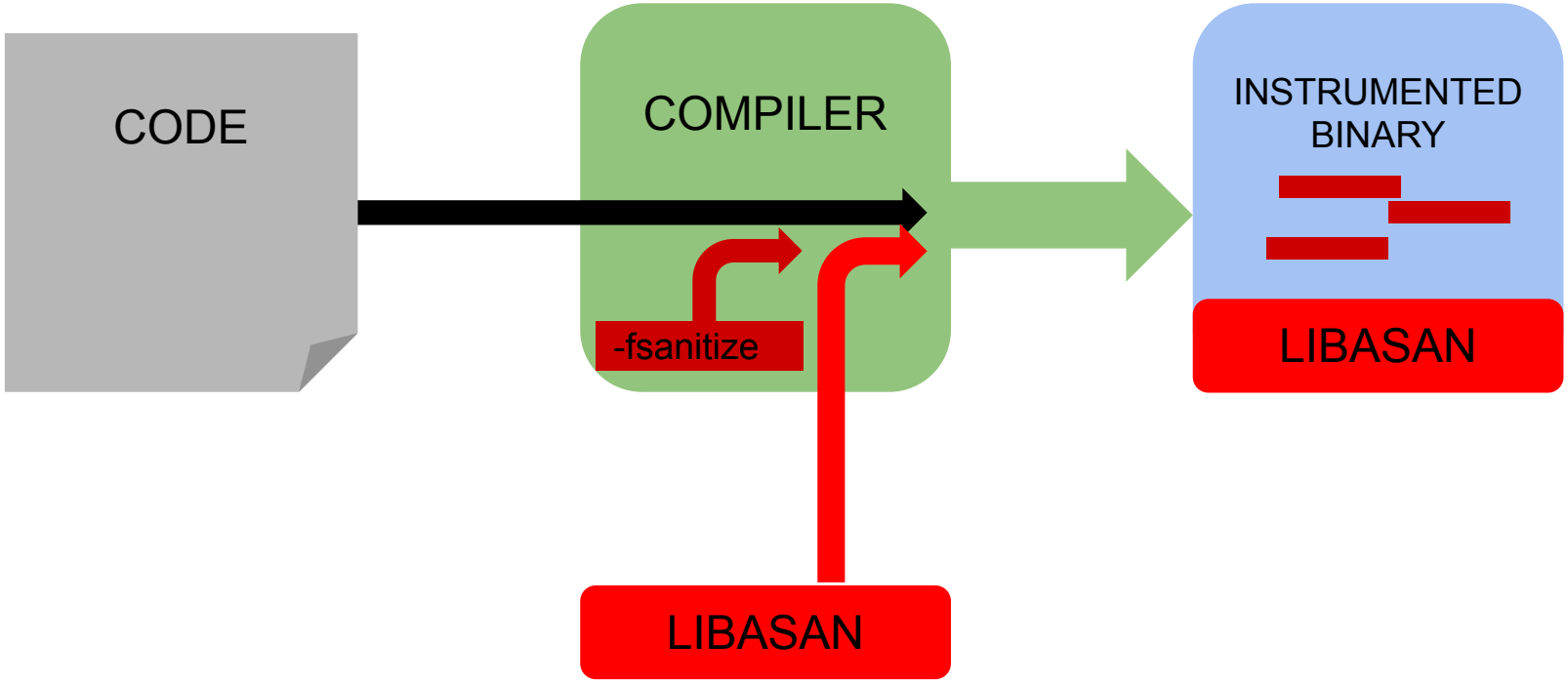
Memory analysis : a dynamic Address Sanitizer

Regular compiling

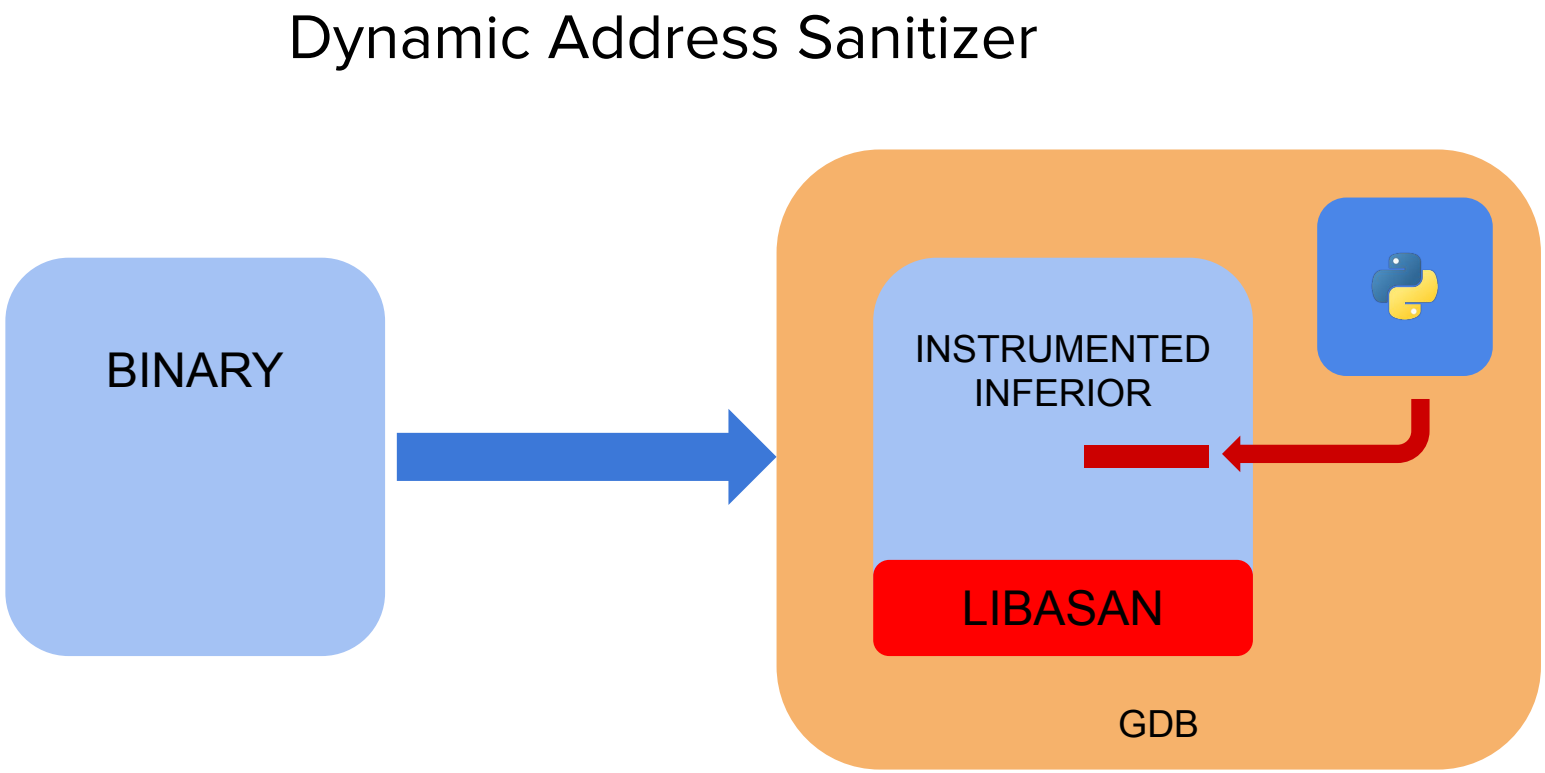


Memory analysis : a dynamic Address Sanitizer

Compile-time Address Sanitizer



Memory analysis : a dynamic Address Sanitizer



Memory analysis : a dynamic Address Sanitizer

Advantages :

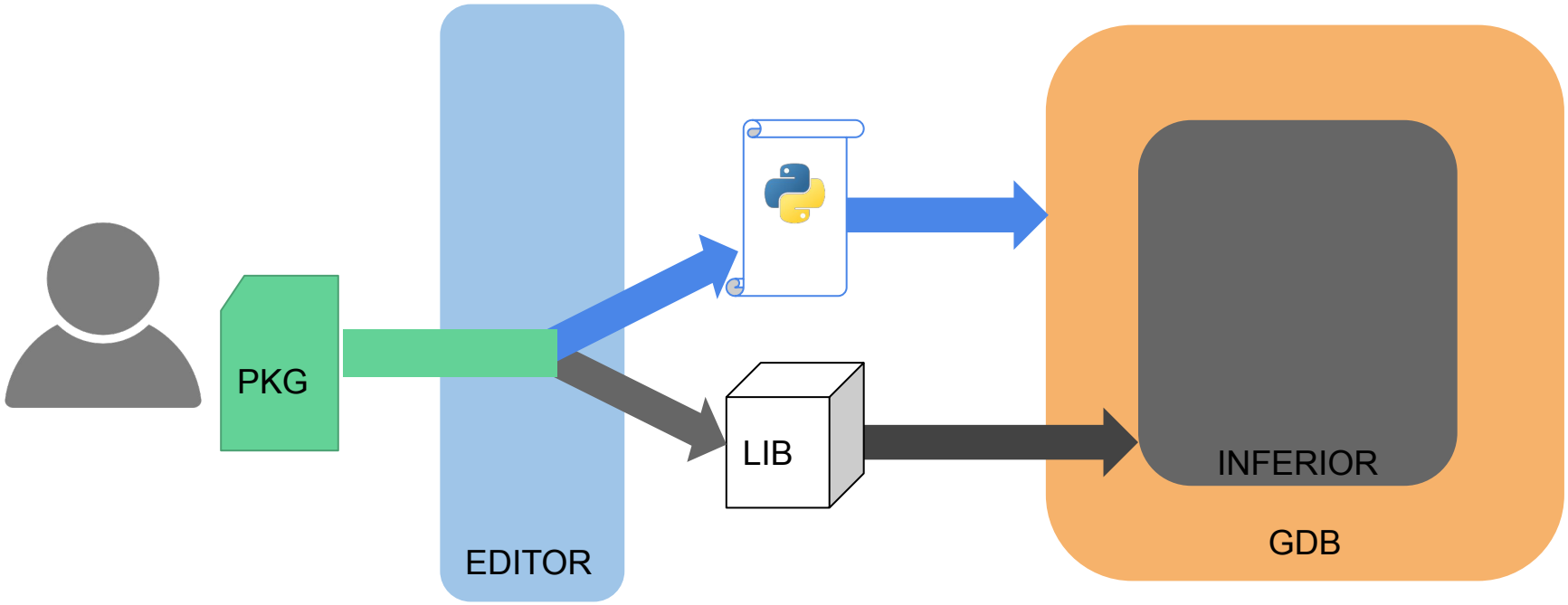
1. No need to have the binary recompiled
2. Can be turned on and off dynamically
3. Can target specific files or lines

Limitations :

1. Can only check for heap faults
2. Needs access to source files
3. It may not be possible to attach it to a running process

Work in progress

A modular platform for instrumentation



Other work

Integrating the patch functionality to GDB upstream

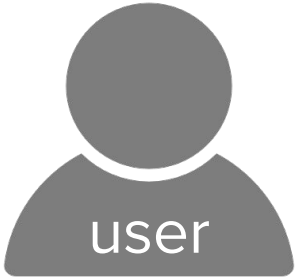
- Two commands : patch and patch asm

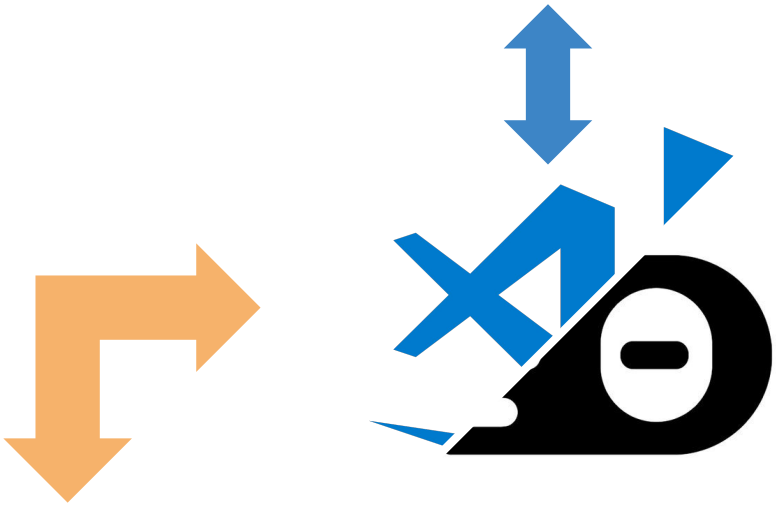
Dynamic tracing using lttng

Looking for input on that !

- Preliminary work by Didier Nadeau : is this still relevant?

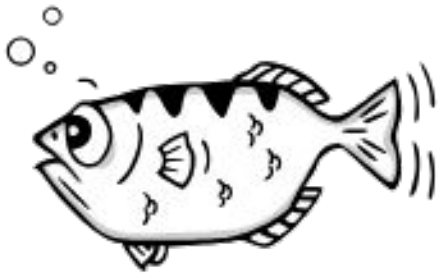
Questions ?





Editor

LSP



GDB

Language Server