Interactive Runtime Verification using a GDB based architecture

08-05-2020

Paul NAERT
Pr Michel DAGENAIS

Summary

- Runtime Verification
- A GDB-based Architecture for RV
- Dynamic instrumentation with GDB
- Use cases
  - Address Sanitizer
  - Data Watch
  - Dynamic C/C++ tracing

Runtime Verification

Runtime Verification : Memory analysis

Valgrind Memcheck

Heap and stack analysis for data corruption, use after free, leaks and use of uninitialized memory

- Fully dynamic
- Significant slowdown (10~100x)
- Instruments all the program
- Cannot attach at runtime
- Command line interface

Address Sanitizer

Heap and stack analysis for data corruption, use after free and leak detection (using Leak Sanitizer)

- Compile-time instrumentation
- Limited slowdown (~2x)
- Instruments all the program
- Cannot attach at runtime
- Command line interface

A GDB-based Architecture for RV

What do current tools lack ?

Features missing

- Targeted instrumentation
- Efficient dynamic instrumentation
- IDE integration
- Attaching to a running program

Features of GDB

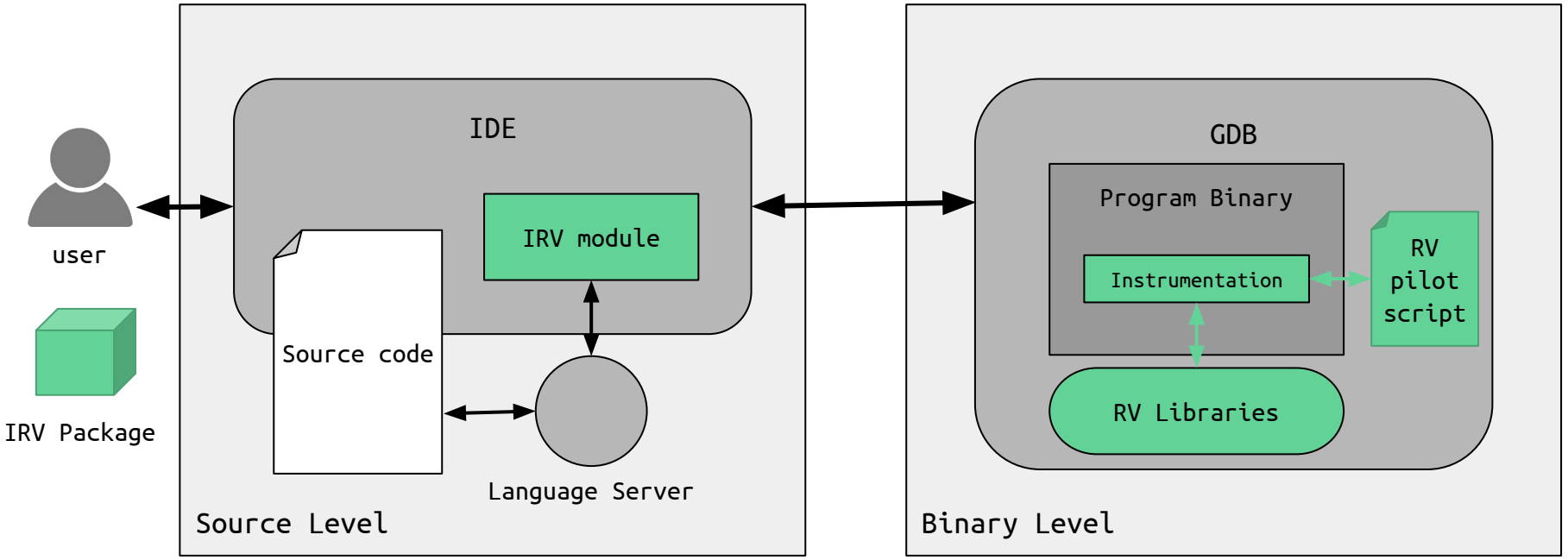

Injection of code at precise locations through breakpoints


Breakpoints are very slow


IDE communication through GDB MI and the Debug Adapter Protocol

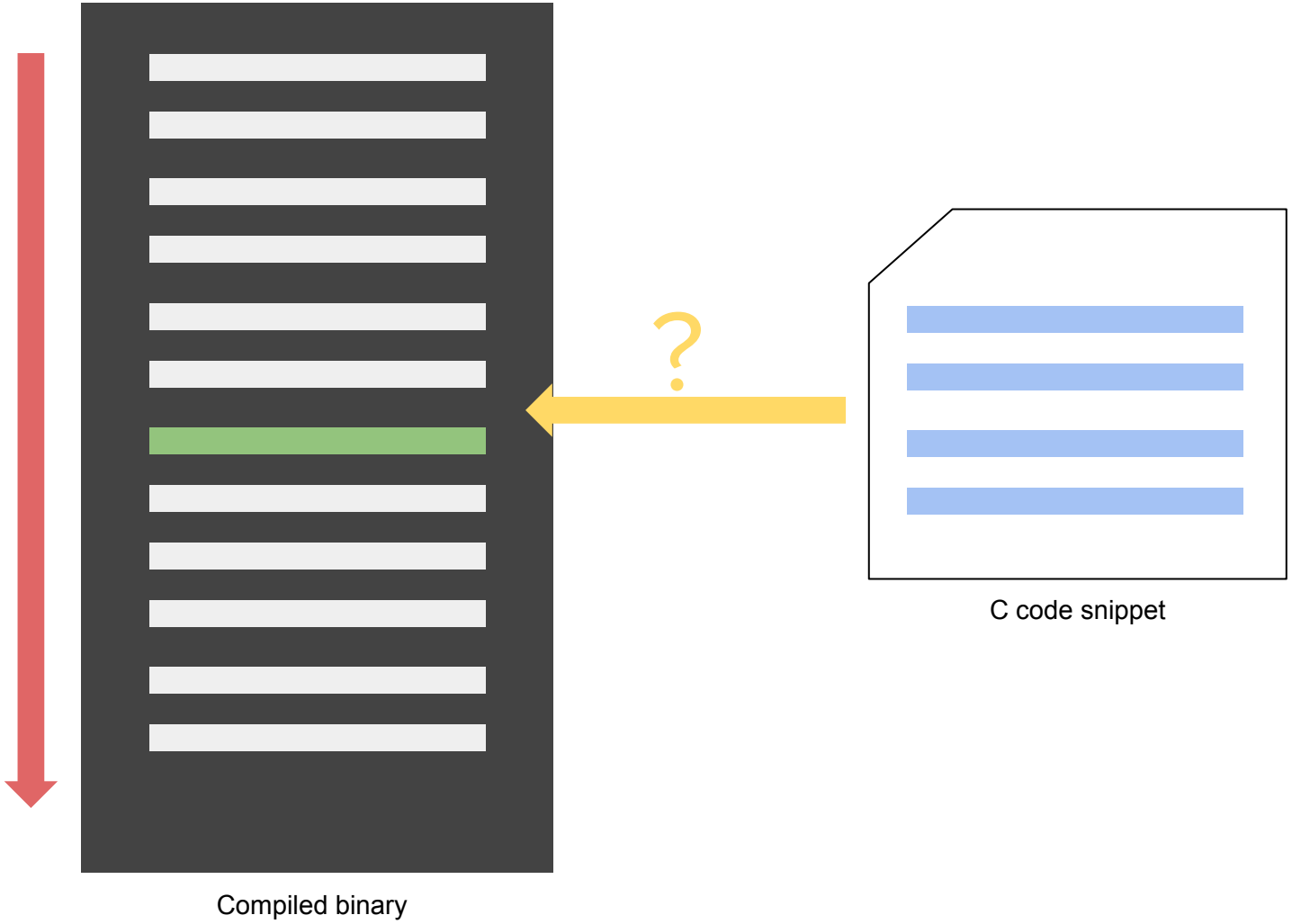

GDB also offers a client-server architecture through *gdbserver*

Our Interactive RV architecture

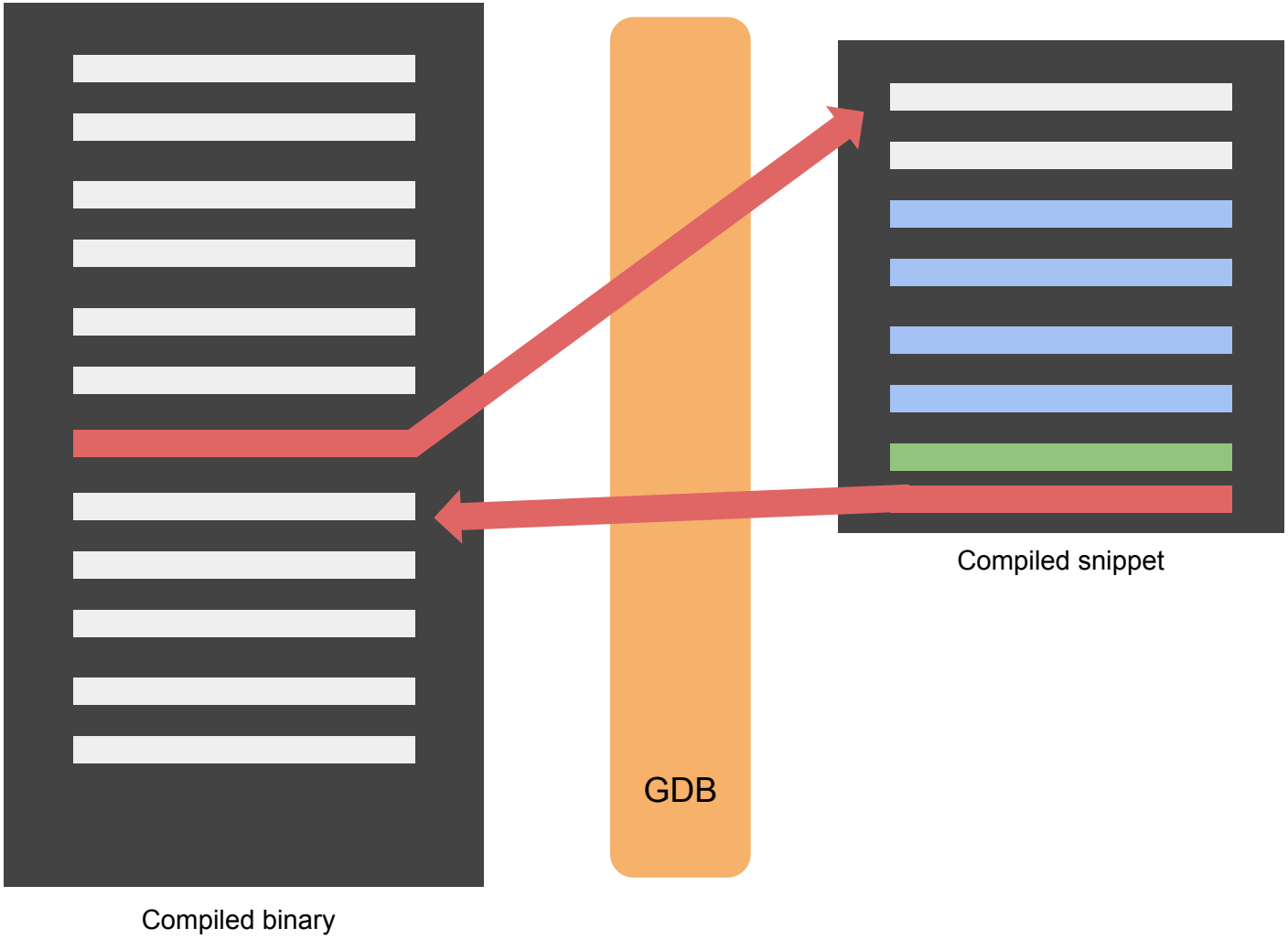


Dynamic instrumentation with GDB

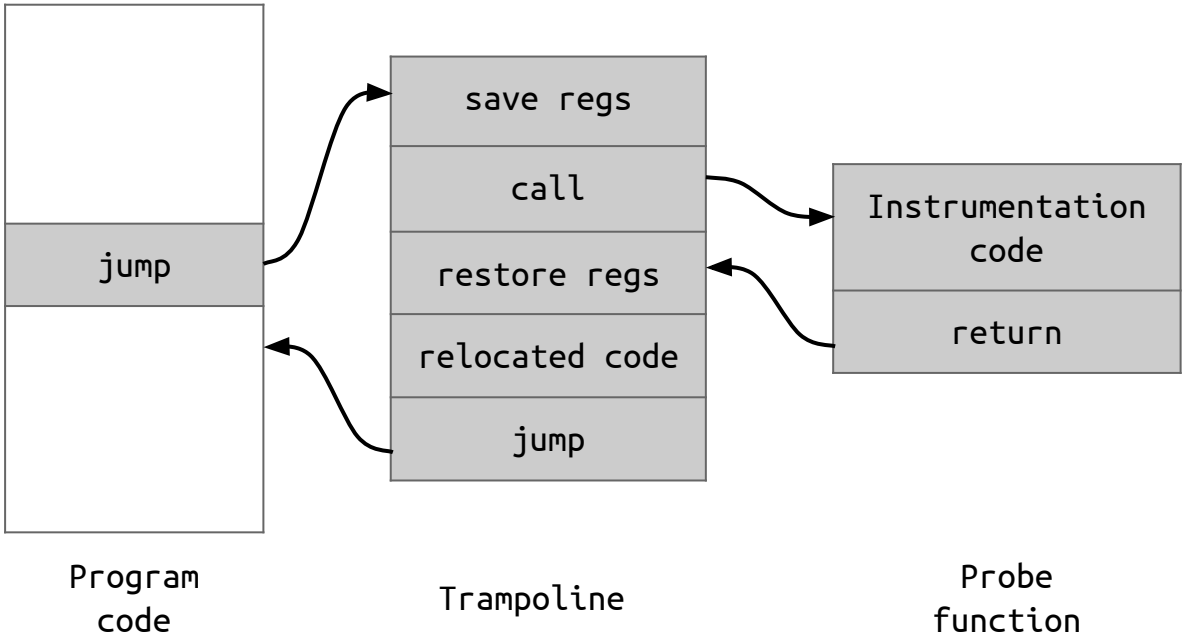
Dynamic code patching



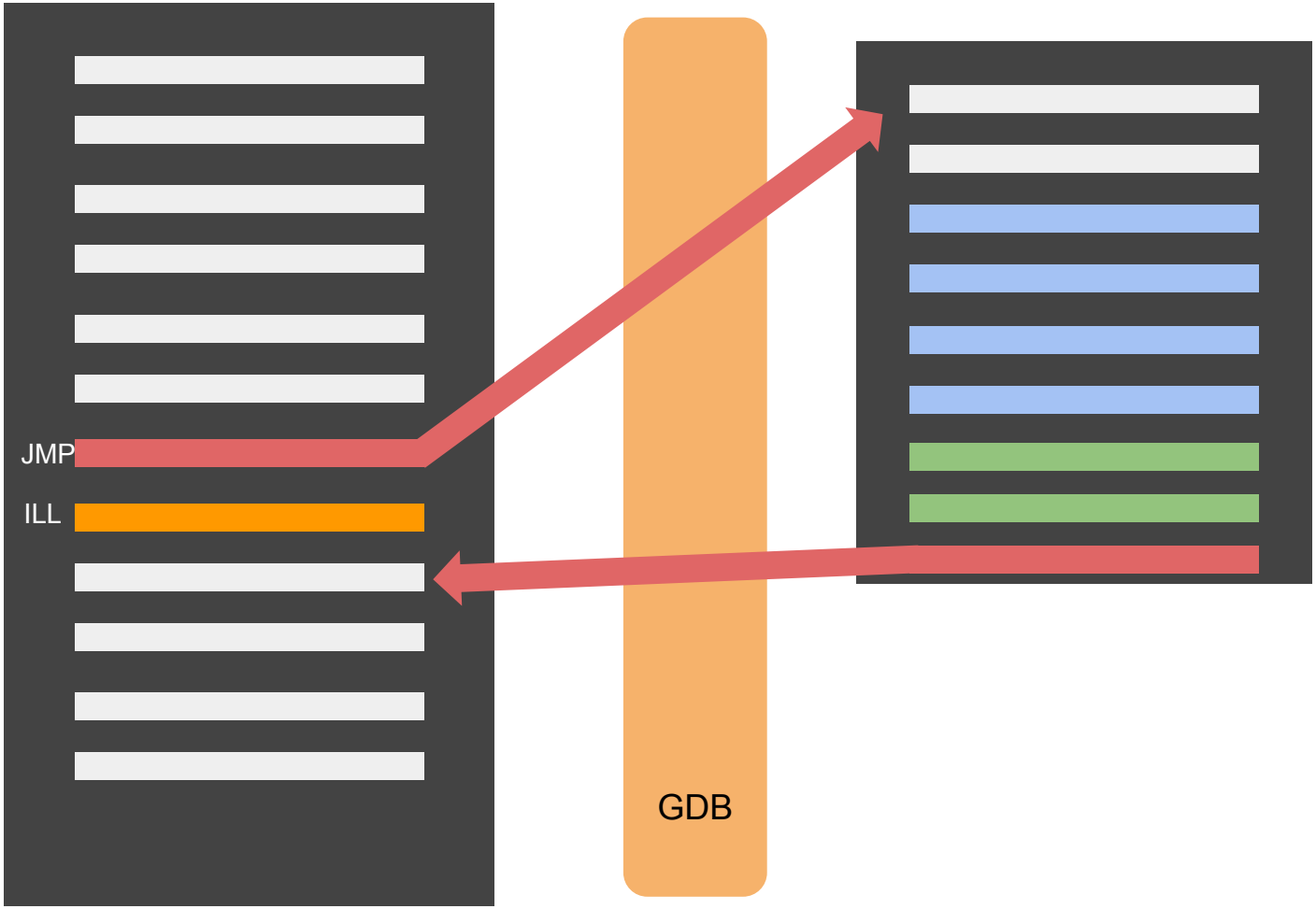
Dynamic code patching



Dynamic code patching - detailed

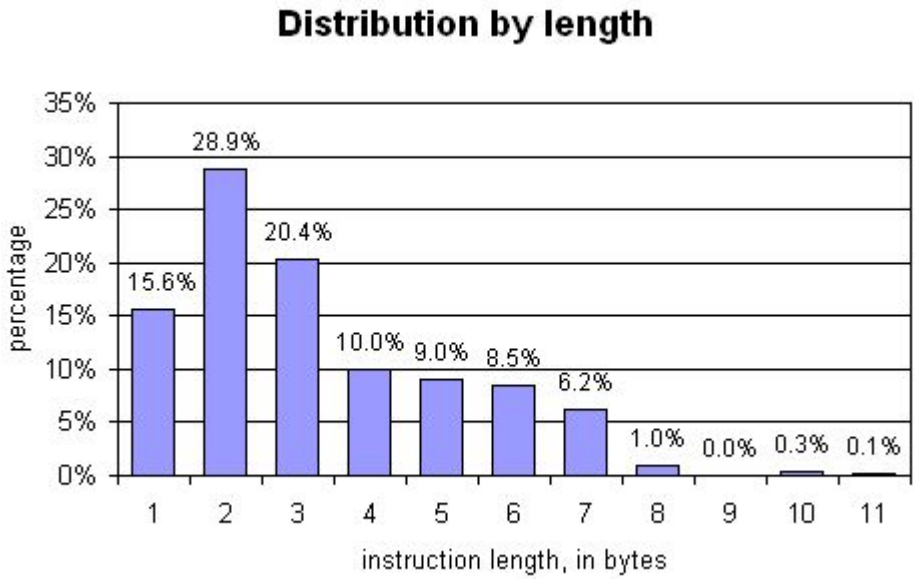


Patching short instructions on x86_64



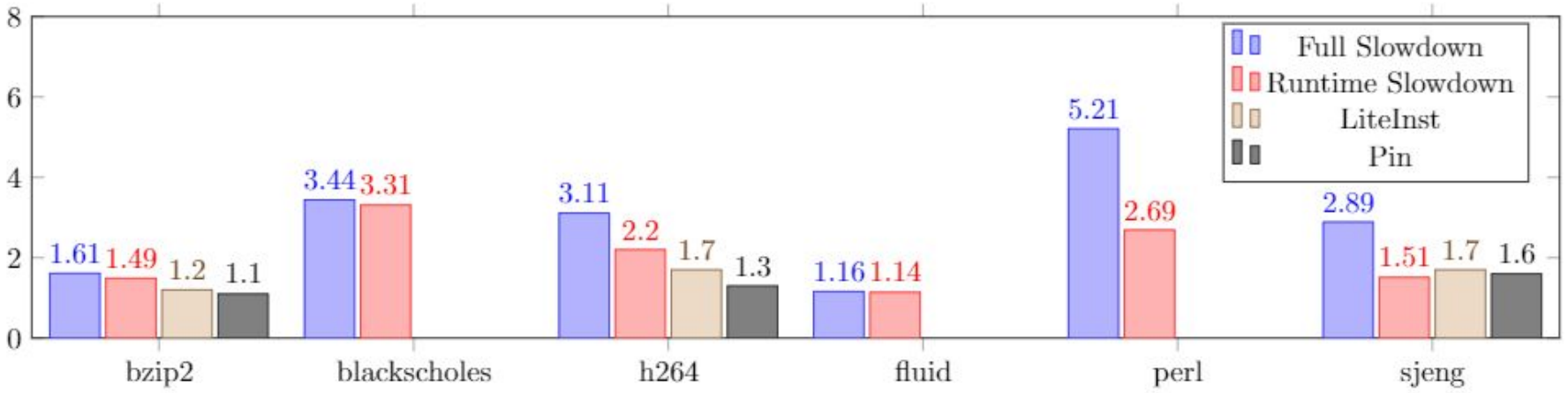
Patching short instructions

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



https://www.strchr.com/x86_machine_code_statistics

Instrumentation Performance on x64



Instrumentation and runtime overhead of a procedure counter on different applications of the SPEC suite, relative to uninstrumented execution time.
Average probe execution time is 50ns, or about 200 CPU cycles

Benchmark	Probes	Mem Overhead	Rel Overhead
bzip2	134	1.39MB	0.6%
blackscholes	36	24KB	0.04%
h264	644	61MB	204%
fluid	108	308KB	0.04%
perl	1996	20.5MB	6.6%
sjeng	188	1.88MB	1.04%

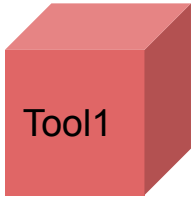
Memory overhead of the procedure counter

Use cases

Expanding a tool with our framework

Tool 1:

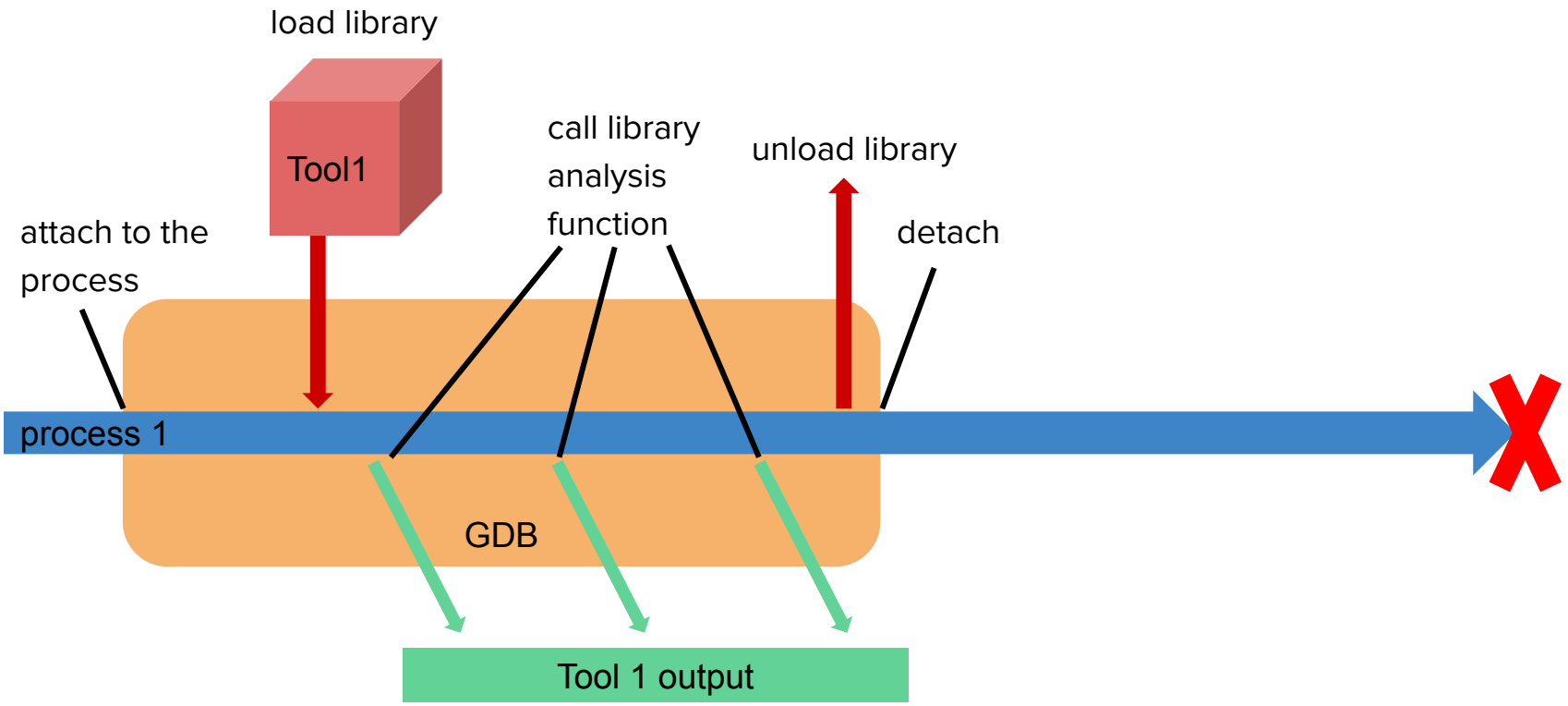
A memory checking library that monitors malloc and free and checks for memory leaks at the end of the program execution.



Process 1:

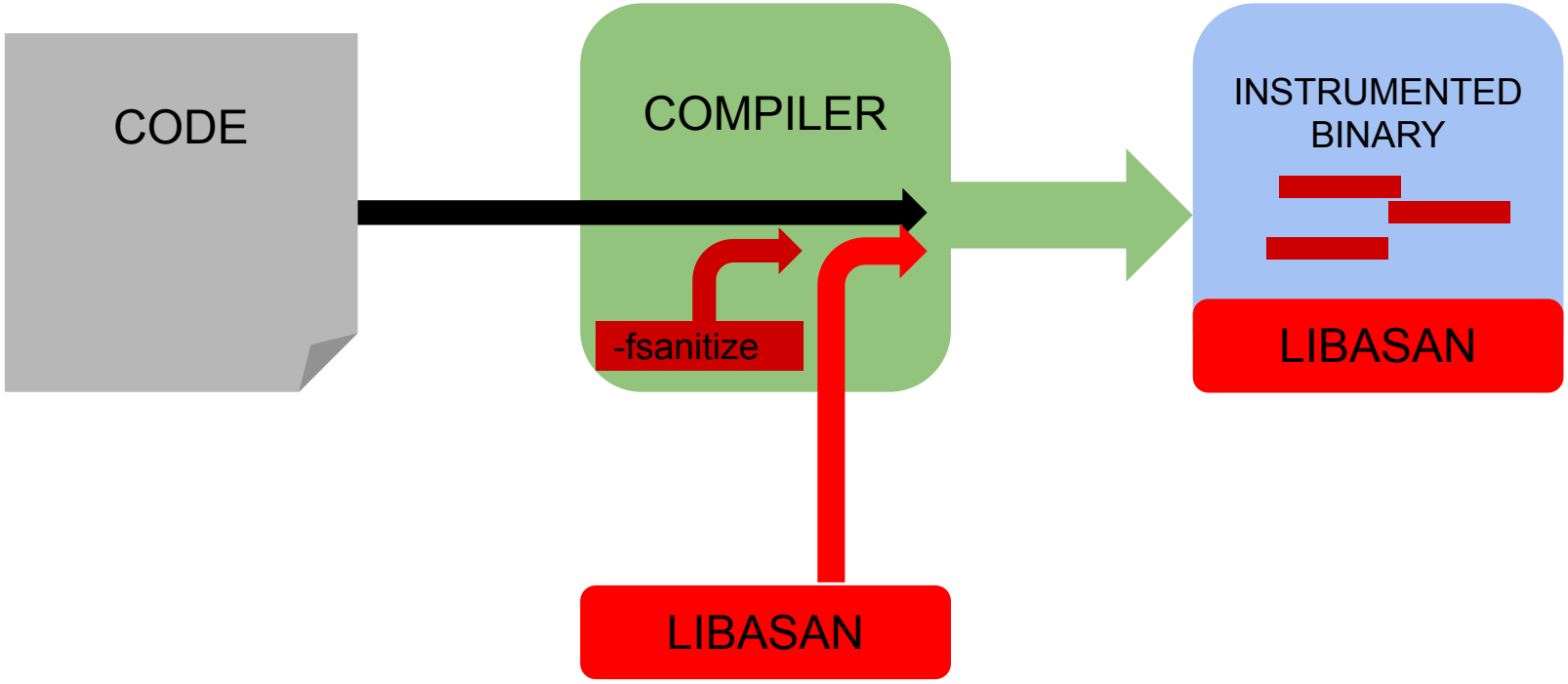
A long running process that sometimes crashes because it runs out of memory.

Expanding a tool with our framework

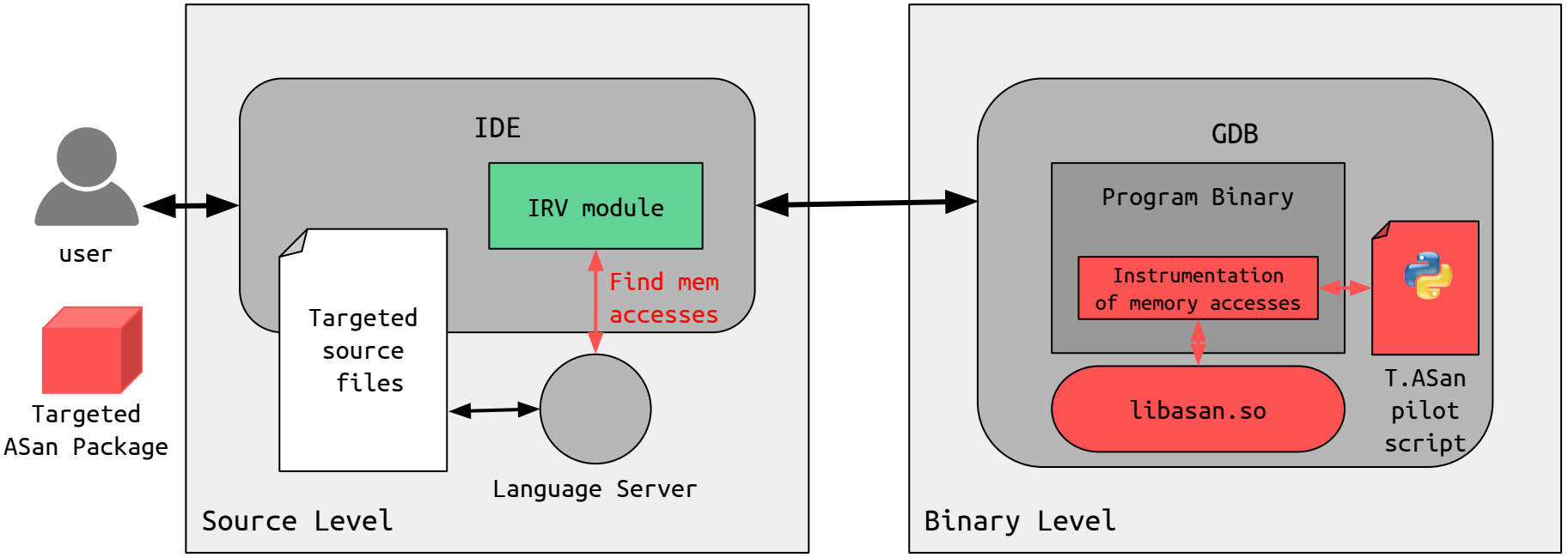


Dynamic Targeted Address Sanitizer

Compile-time Address Sanitizer

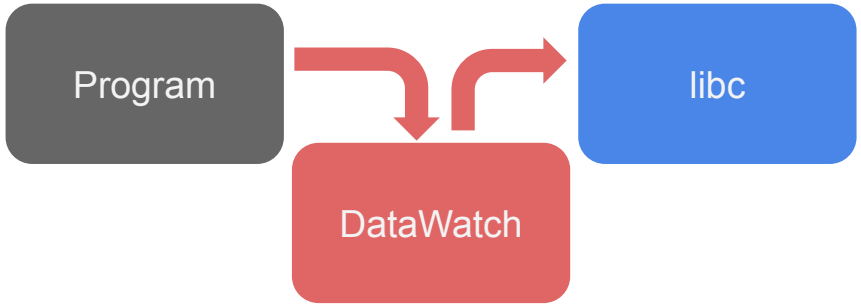


Dynamic Targeted Address Sanitizer

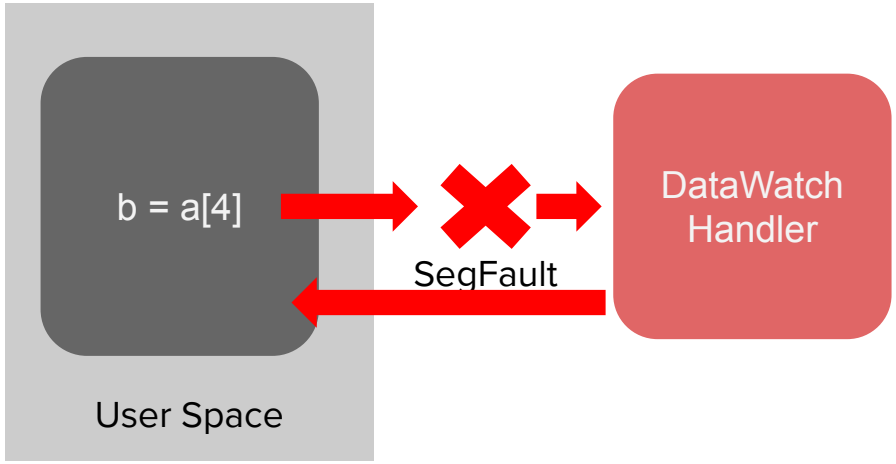


User-space only Data Watch

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.

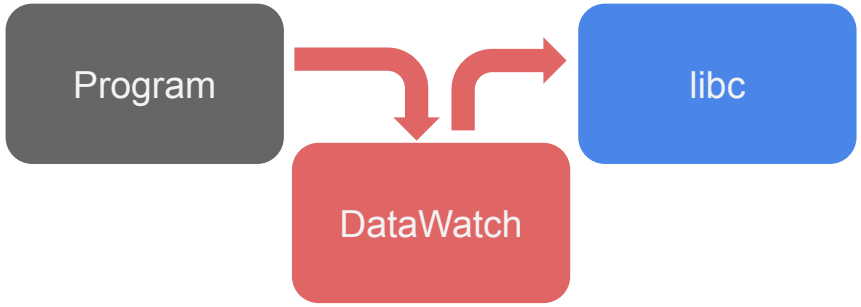


2. Each pointer resolution now raises a segmentation fault, which is handled by DataWatch. If the dereference is within bounds, it corrects the address and sends back the value.

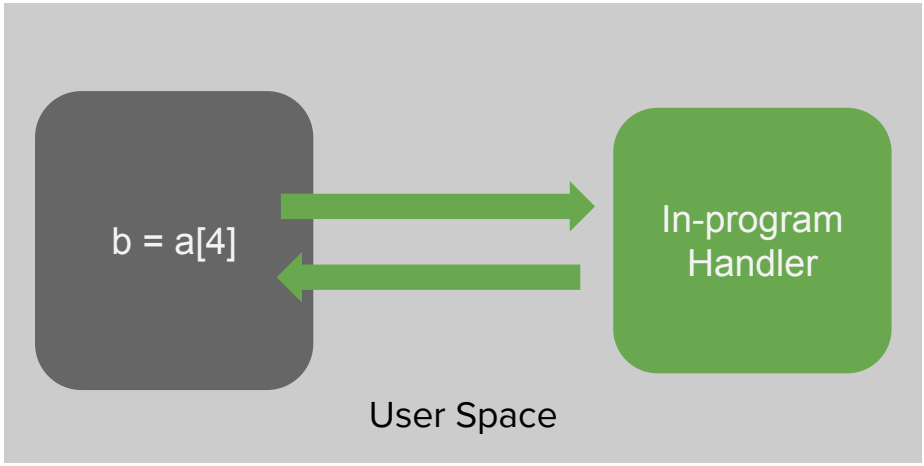


User-space only Data Watch

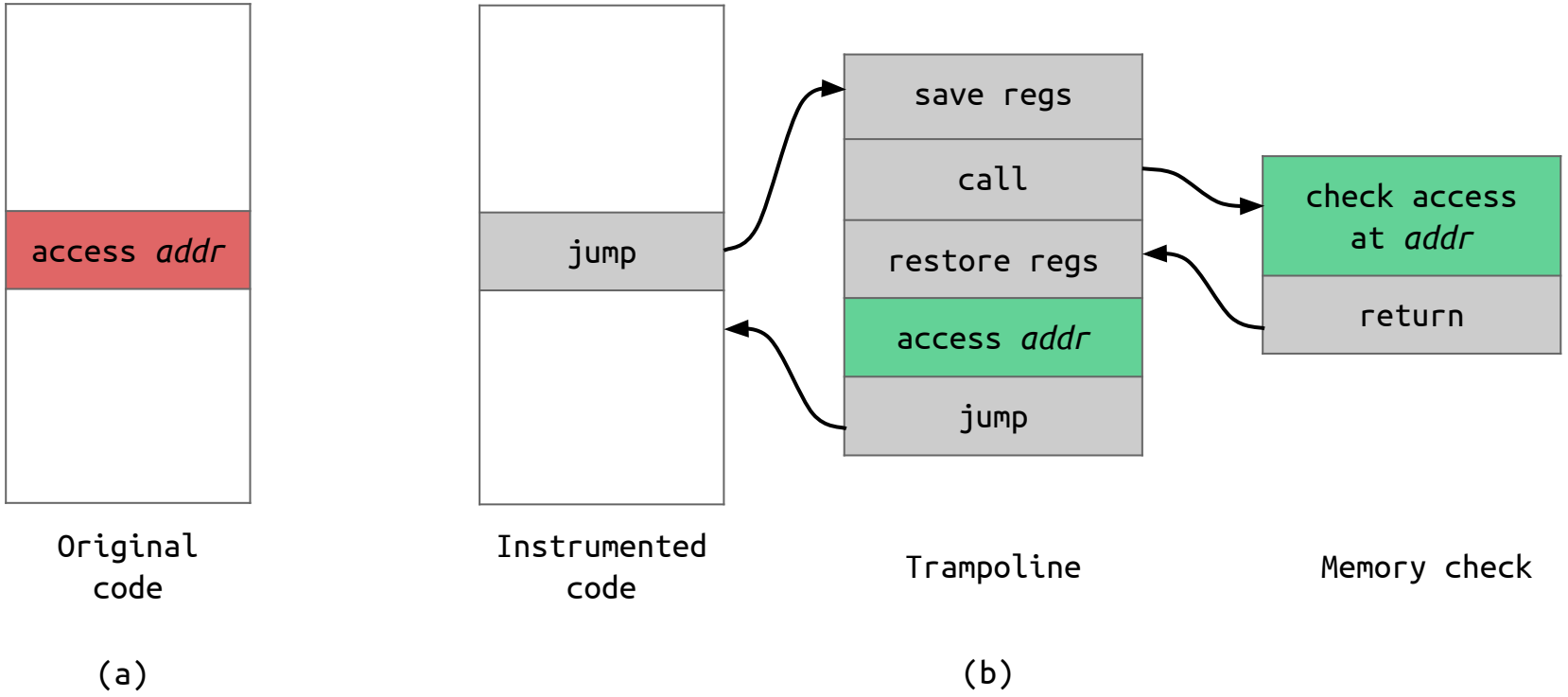
1. Override malloc() and free(). malloc() now adds information in the most significant bits of the address, and stores what has been allocated and where.



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.



User-space only Data Watch



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.
   This would need a Kernel module

Limitations :

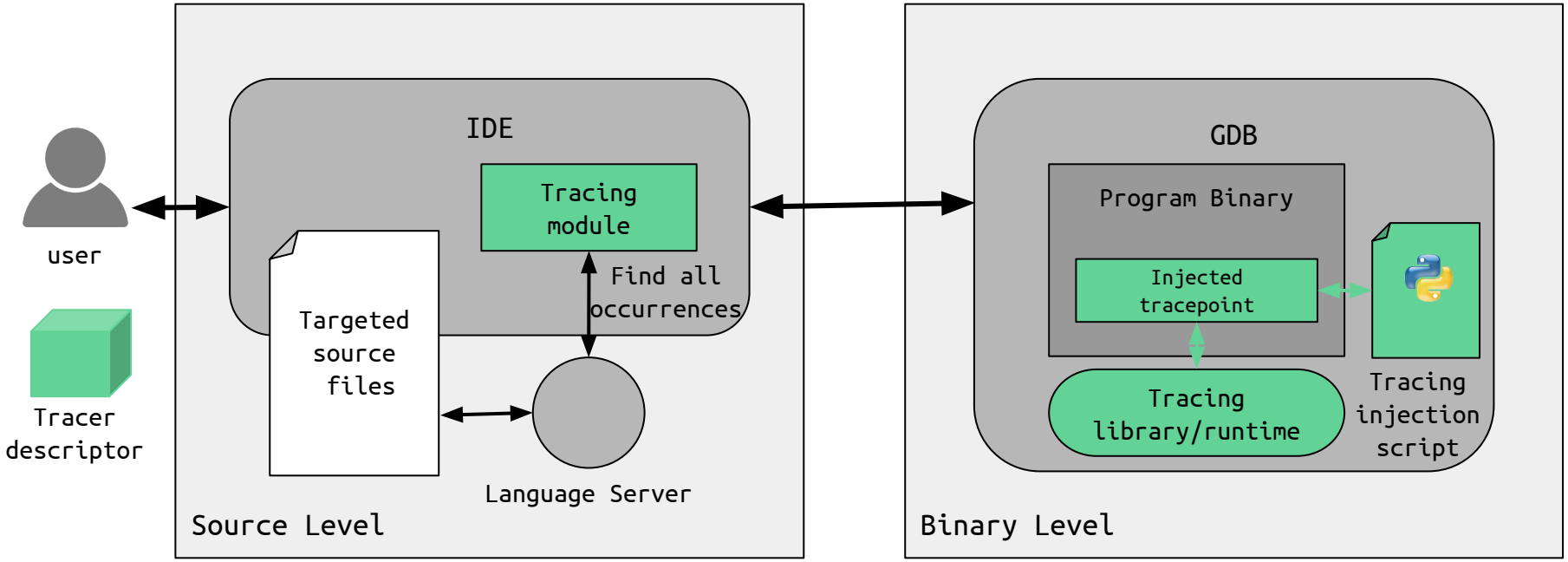
1. Overhead can be significant in libc : giving an invalid address to strcpy will cause a large number of illegal instructions to be hit.
2. No complete override of malloc and free : memory allocated outside of the target range will not be checked.
3. No verification for data allocated on the stack.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

Dynamic C/C++ tracing

- Information about a process execution

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

Dynamic C/C++ tracing

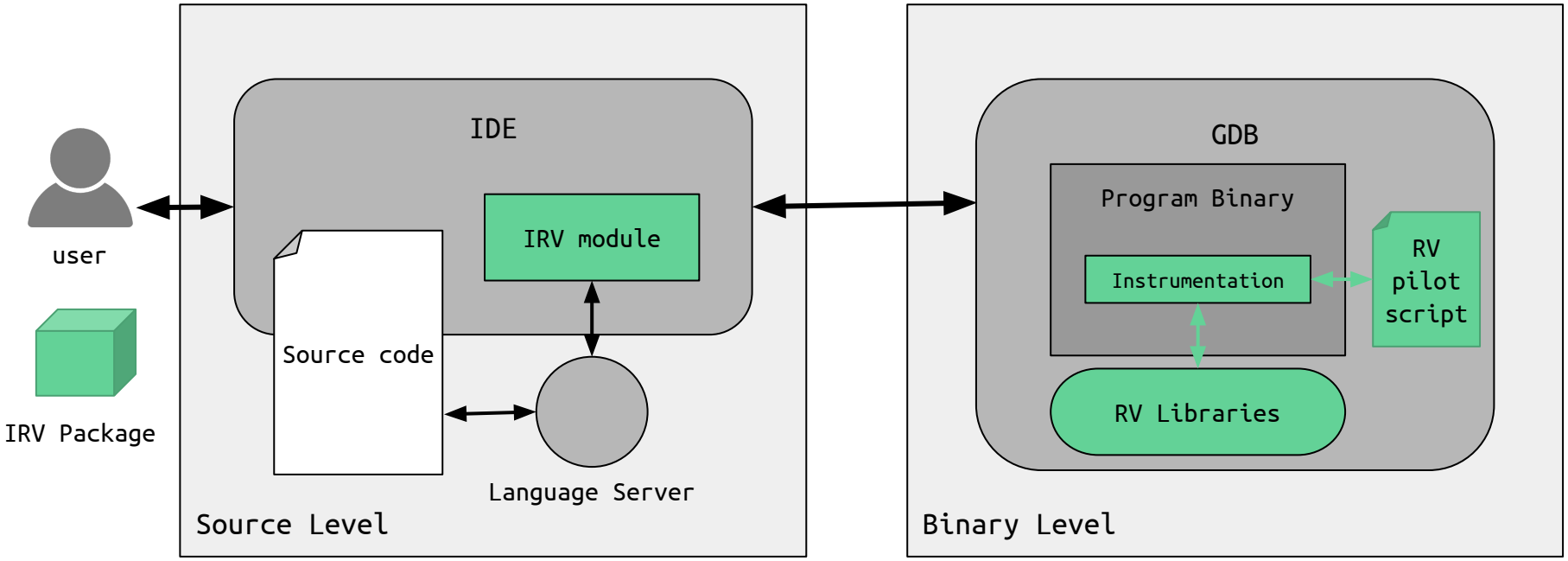


Using the minitrace library :

85ns average overhead per tracepoint

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

Conclusion


user
IRV Package
IDE
IRV module
Source code
Language Server
Source Level
GDB
Program Binary
Instrumentation
RV
pilot
script
RV Libraries
Binary Level