

GDB as a service

06-05-19

Paul NAERT
Pr Michel DAGENAIS

Summary

- ❑ Introduction
 - ❑ What is GDB ?
 - ❑ The Language Server Protocol

- ❑ GDB as a 'binary execution server'

- ❑ Applications
 - ❑ Dynamic C/C++ tracing
 - ❑ Memory analysis

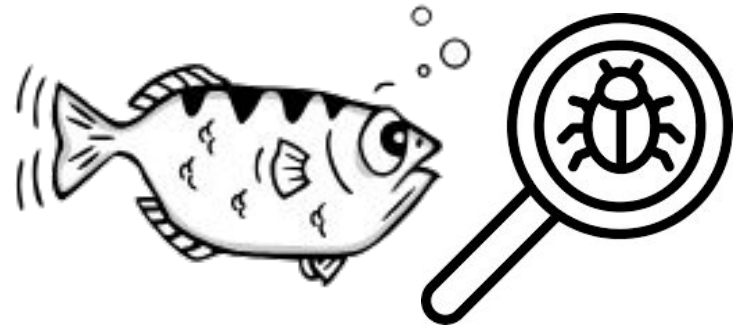
- ❑ Conclusion

Introduction

What does the GNU Debugger do?

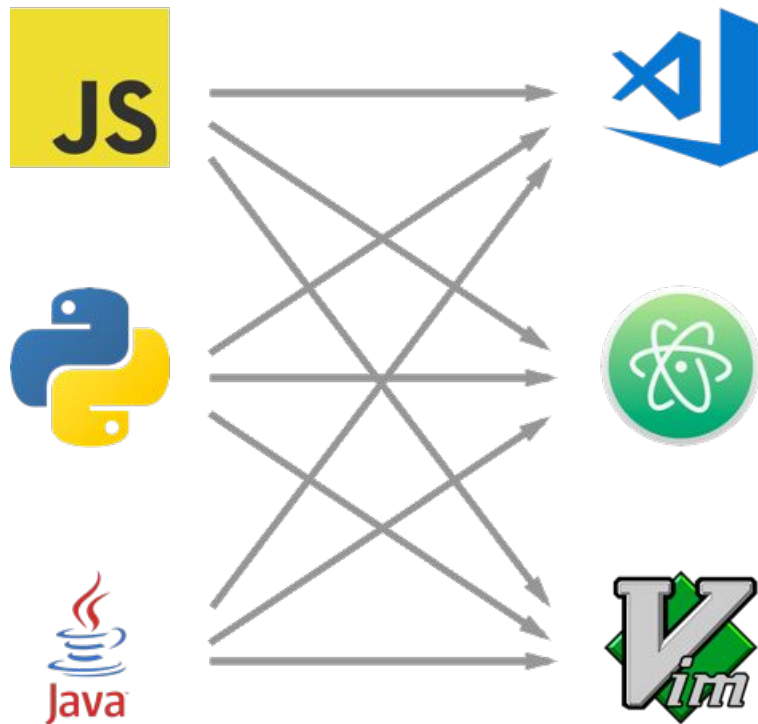
Observe what is going on during a program execution.

- Execute a program line by line
- Display variable and register values
- Set breakpoints



The Language Server Protocol

NO LSP



LSP

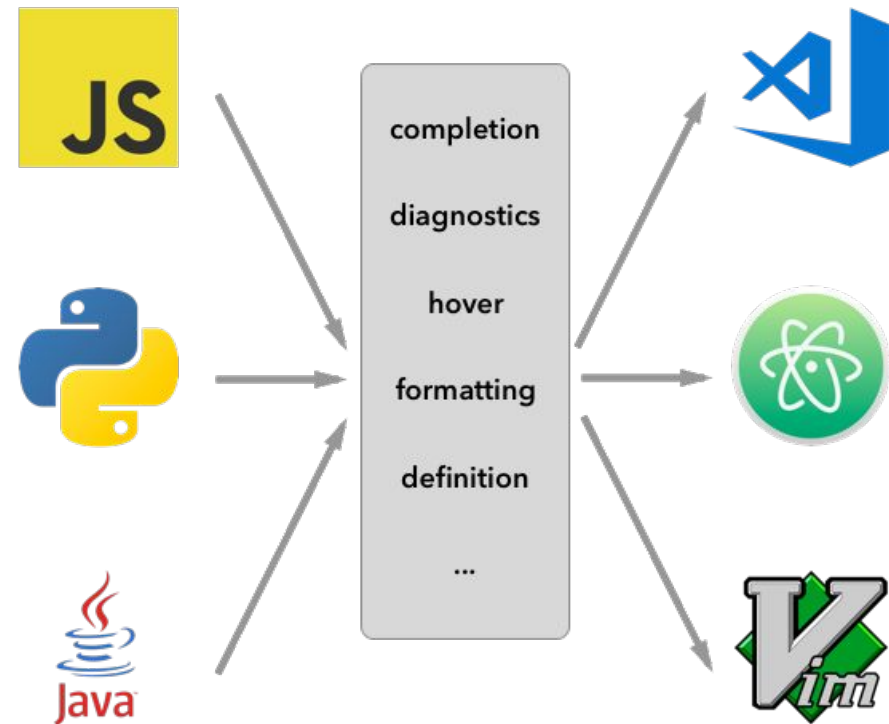
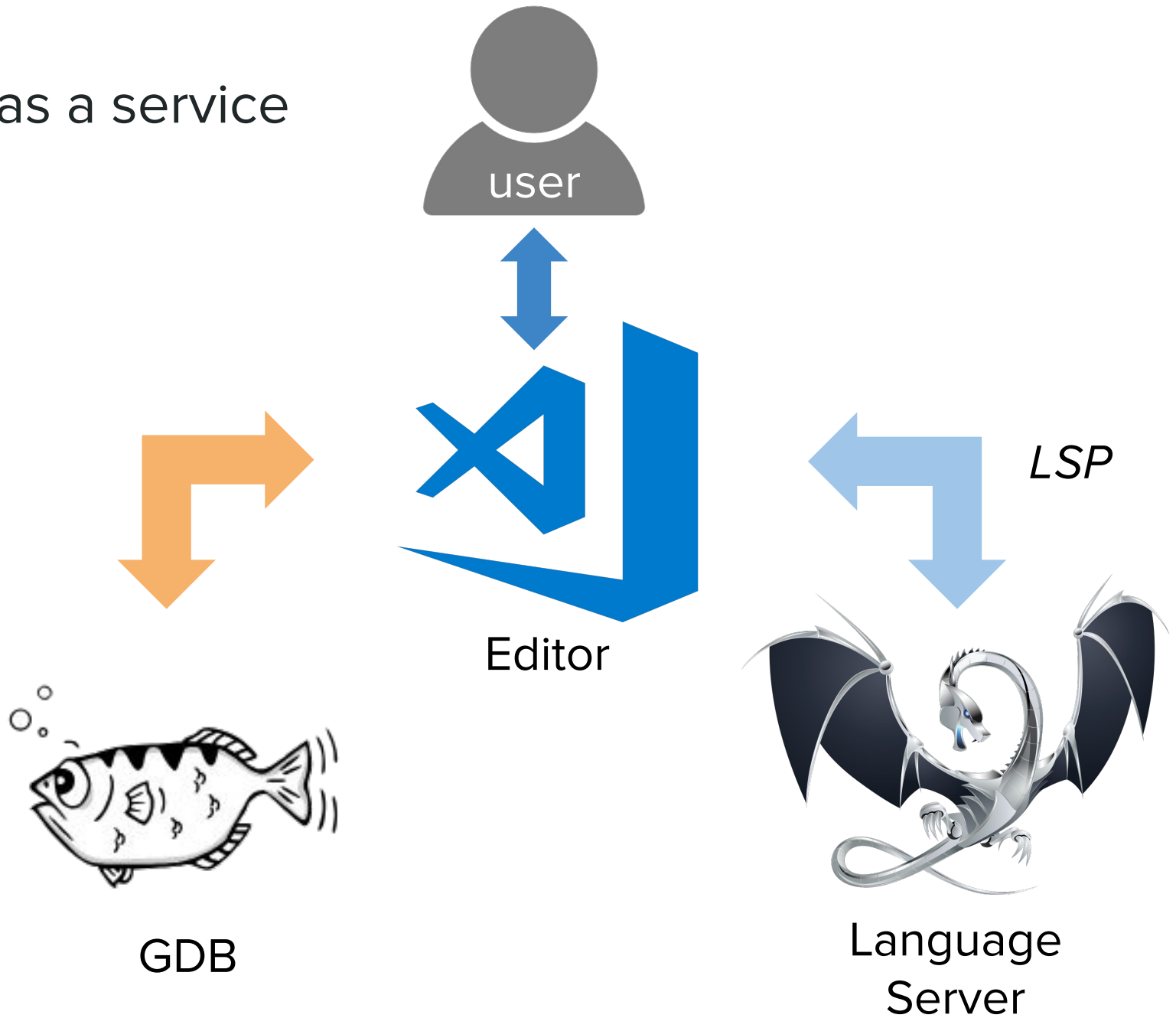


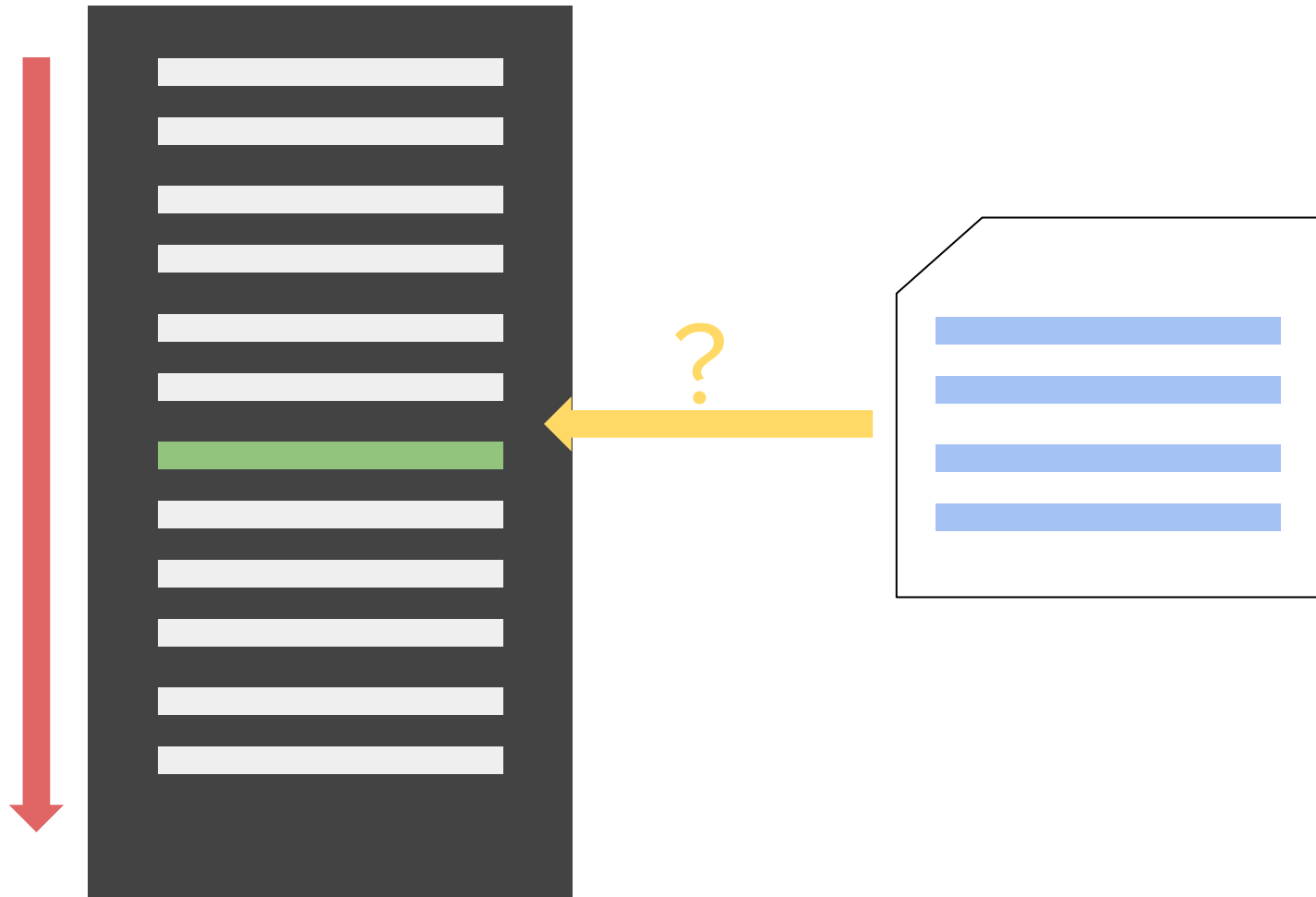
Illustration from VSCode website

GDB as a 'binary execution server'

GDB as a service



Dynamic code insertion



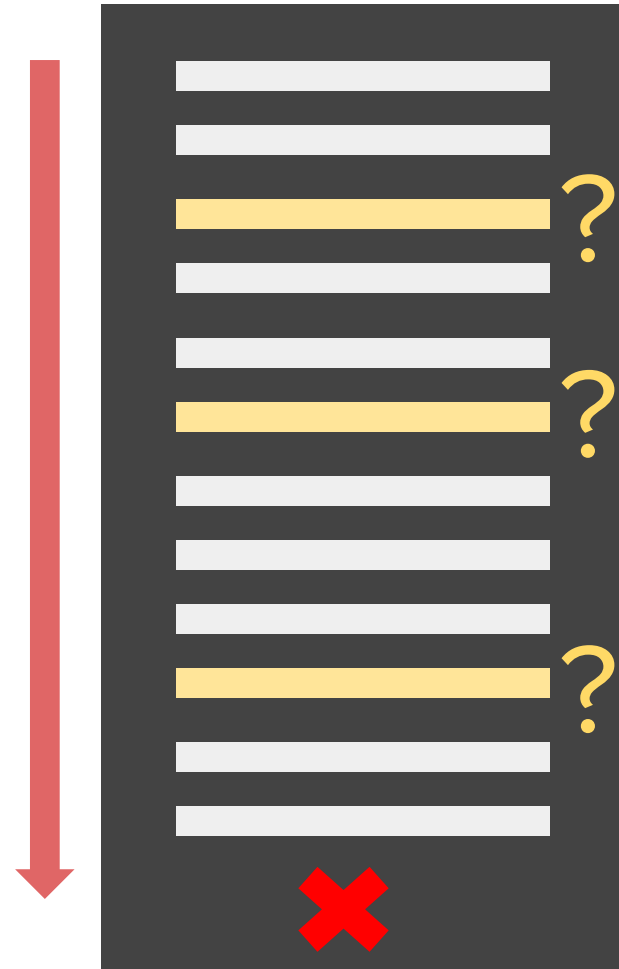
Dynamic code insertion



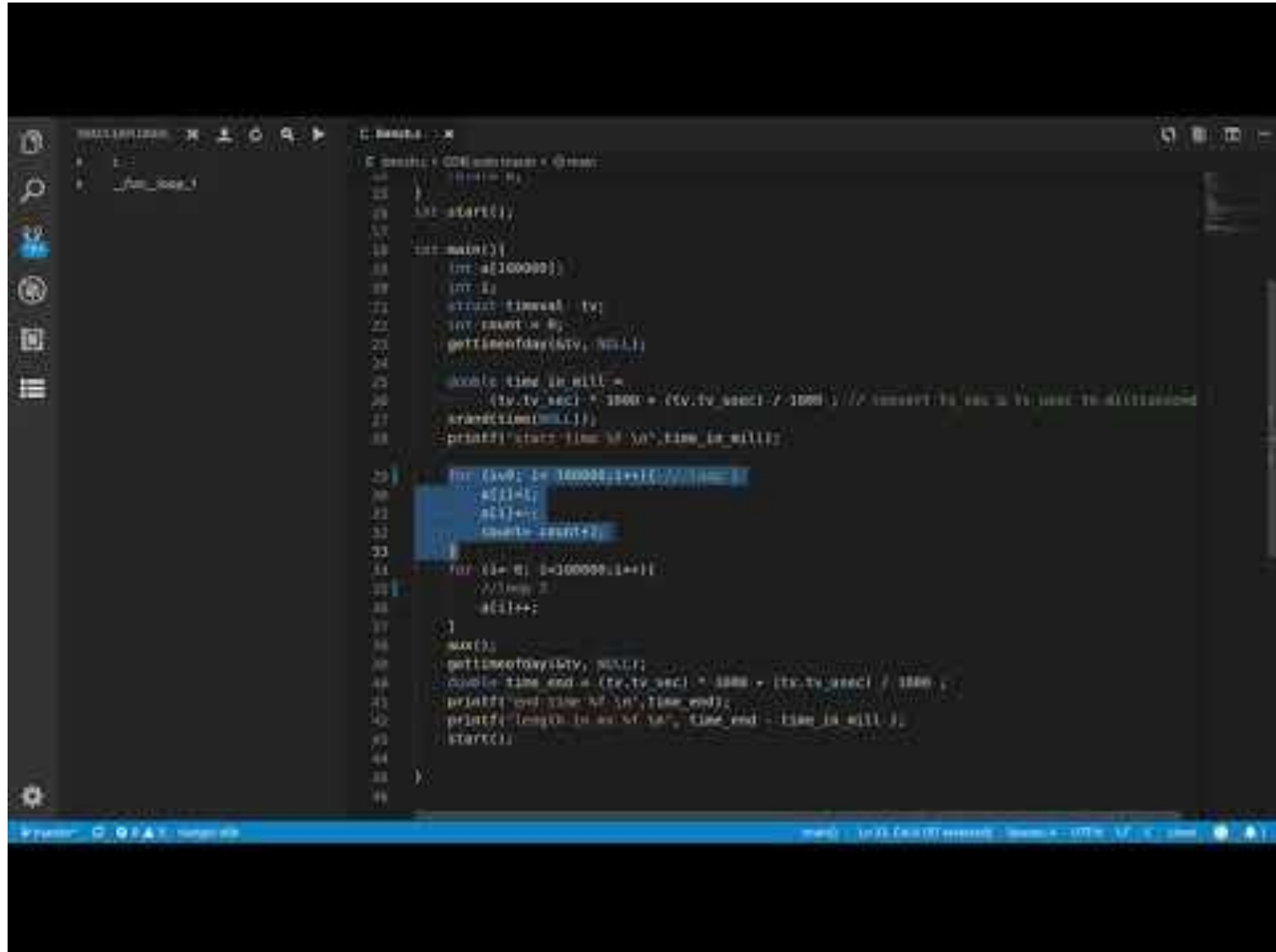
Applications

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 : editor integration

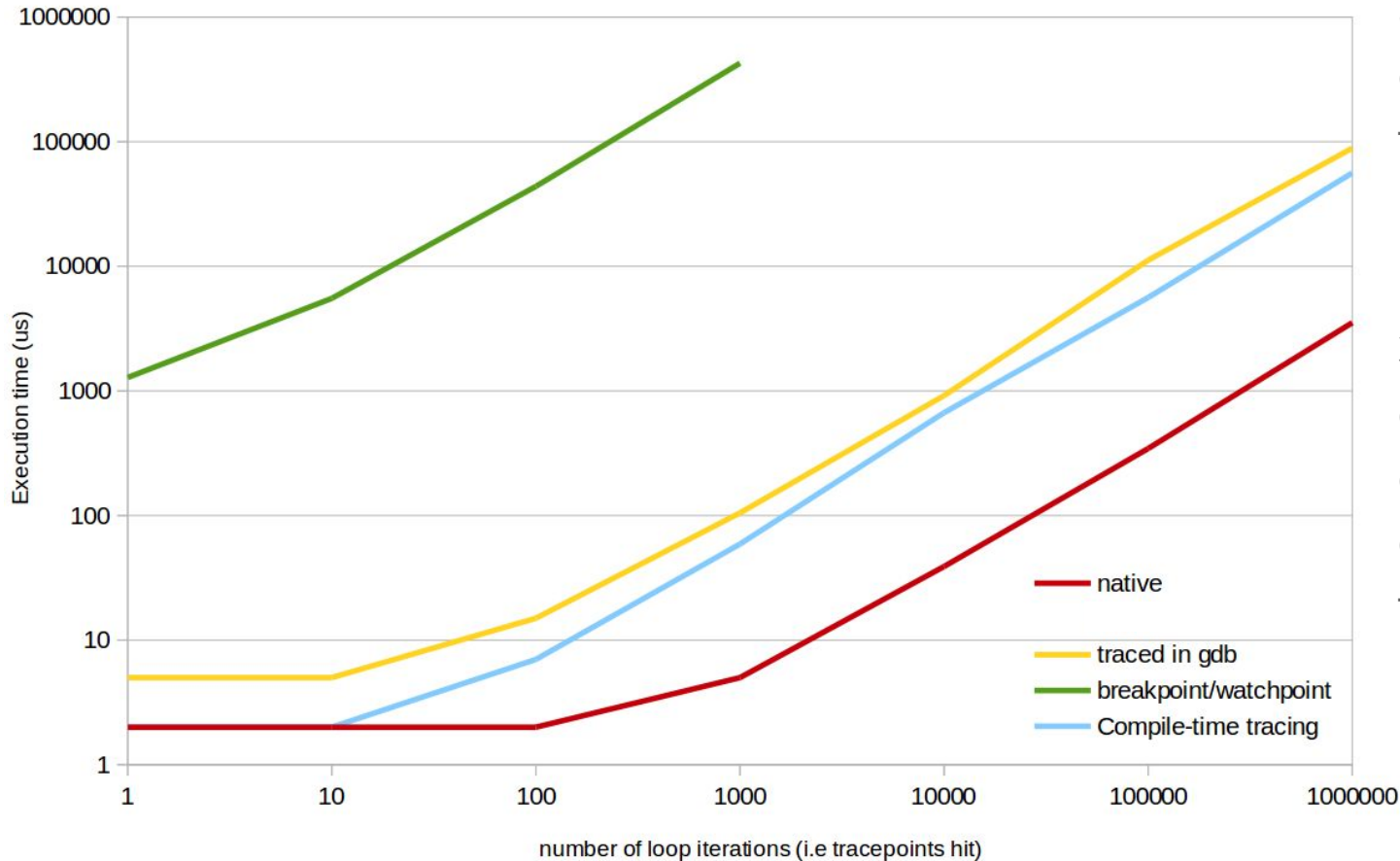


The image shows a screenshot of a code editor with a dark theme. The editor displays C++ code for a program that benchmarks a loop. The code includes headers for `time.h` and `sys/time.h`. It defines a `timeval` struct and uses `gettimeofday` to measure time. The main function includes a loop that increments an array `a` and a counter `count`. The code is annotated with tracing macros: `START()` at the beginning and end of the program, `STOP()` at the end of the loop, and `TRACE()` at the end of each iteration. The `TRACE()` macro is highlighted in blue. The code also includes comments for calculating time in milliseconds and microseconds.

```
1 #include <time.h>
2 #include <sys/time.h>
3
4 struct timeval {
5     long tv_sec;
6     long tv_usec;
7 };
8
9 int main() {
10     int a[100000];
11     int i;
12     struct timeval tv;
13     int count = 0;
14     gettimeofday(&tv, NULL);
15
16     double time_in_milli =
17         (tv.tv_sec * 1000 + (tv.tv_usec / 1000)) // convert to sec to use to gettimeofday
18     gettimeofday(NULL);
19     printf("start time is %0^", time_in_milli);
20
21     for (i=0; i< 100000; i++) { // loop 1
22         a[i]=i;
23         a[i]=a[i];
24         count = count+1;
25     }
26     for (i= 0; i<100000; i++)
27         // loop 2
28         a[i]++;
29     }
30     gettimeofday(&tv, NULL);
31     double time_end = (tv.tv_sec * 1000 + (tv.tv_usec / 1000));
32     printf("end time is %0^", time_end);
33     printf("length is %0^", count);
34     printf("time_end - time_in_milli");
35     START();
36 }
```

Dynamic C/C++ tracing

Tracing overhead on a simple program



85ns average overhead per tracepoint

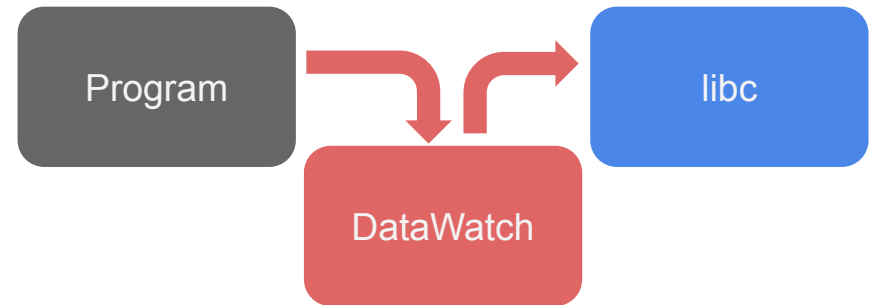
30ns average overhead compared to compile-time tracing

Memory leak and corruption analysis

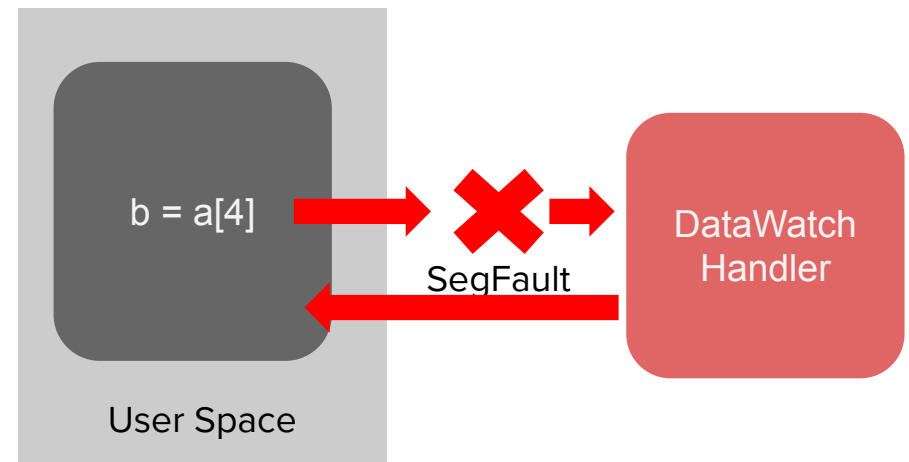
- Allocated memory is never freed
- Memory is corrupted due to out of bounds access (buffer overflow ...)
- Tools exist : Valgrind, Address Sanitizer, Data Watch ... but they are slow or they need the binary recompiled.

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.

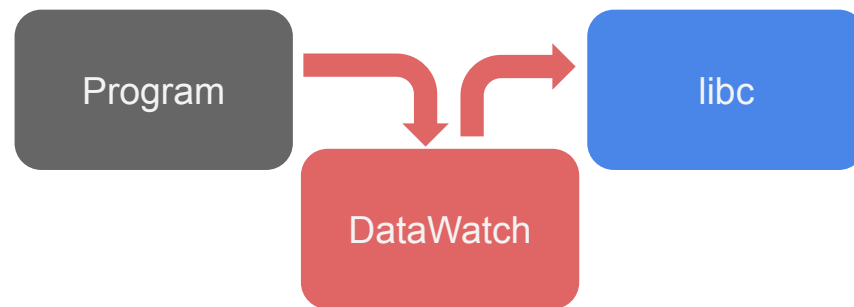


2. Each pointer resolution now raises a segmentation fault, which is handled by DataWatch. If the deallocation is within bounds, it corrects the address and sends back the value.

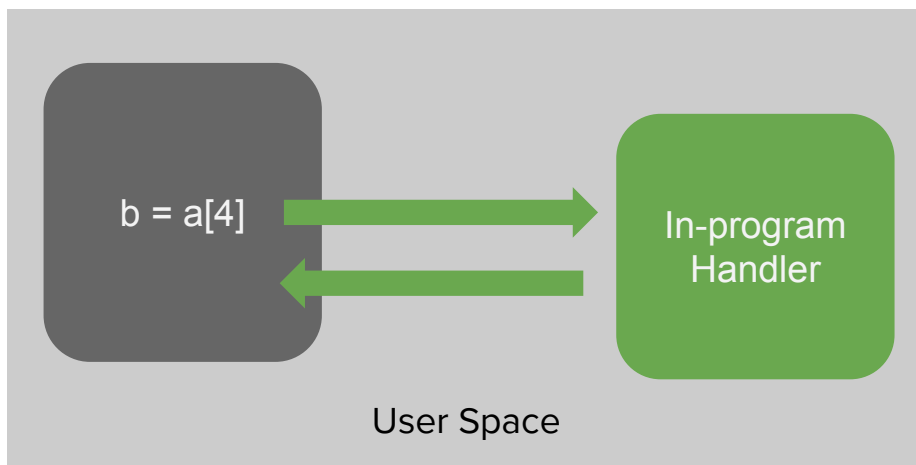


Memory analysis : a user-space only Data Watch

1. Impersonates libc to 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 deallocation is within bounds, it corrects the address and sends back the value.~~
The resolution is corrected in the program without any transition to kernel space.



A first prototype

1. Preload the library using LD_PRELOAD and insert initialization function at the start of the program.
2. Run the source file through the clang to get the AST dump.
3. In the AST find calls to malloc and free and replace the call addresses in the binary using GDB.
4. Using the AST, find all pointer dereferences (*, [], ->), and insert a call to the library to check and correct the pointer address.

```
callq 0x401070 <malloc@plt>
```

3.

```
movabs $0x7ffff7ff21b7,%rax  
jmpq   *%rax <malloc_wrapper>
```

```
array[j][i] = 1;
```

4.

```
*(long*)memory_access(*(long*)memory_access(array+j)+i) = 1;
```

A first prototype

Pros :

1. It should be faster than vanilla DataWatch, as it does not involve switching to kernel space.
This has yet to be verified
2. It can target only a specific part of a program.
3. Can work in conjunction with Data Watch if pointers are shared outside of the targeted area.
4. Can be attached to a running binary, although it will not check already allocated memory.
Not yet implemented

Cons :

1. You need the debug symbols for the binary and you must have the source code for the parts being instrumented.
2. GDB overhead
3. No complete override of malloc and free : memory allocated outside of the target range will not be checked.
4. I was not able to handle some C syntax (*p++ for instance)
5. No verification for data allocated on the stack.

Conclusion

GDB can be more than a debugger via integration with text editors

Applications include :

- Dynamic low-cost tracing

- Efficient and targeted memory analysis

Questions ?

