Dissecting Google's AddressSanitizer (ASAN)

Vishal Juneja, Computer Science (Cybersecurity)

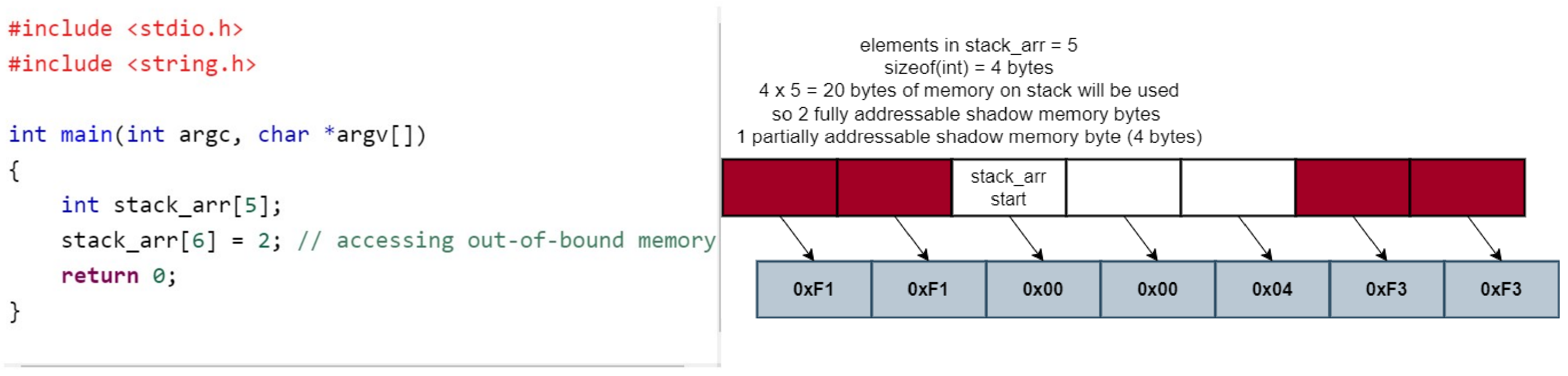
Mentor: Yan Shoshitaishvili, Associate Professor, School of Computing and Augmented Intelligence

Introduction

AddressSanitizer (ASAN) is an automatic error detection tool made by Google. It is used to find bugs such as Use-After-Free, Out-Of-Bound accesses to heap, stack and global objects etc.

Observations

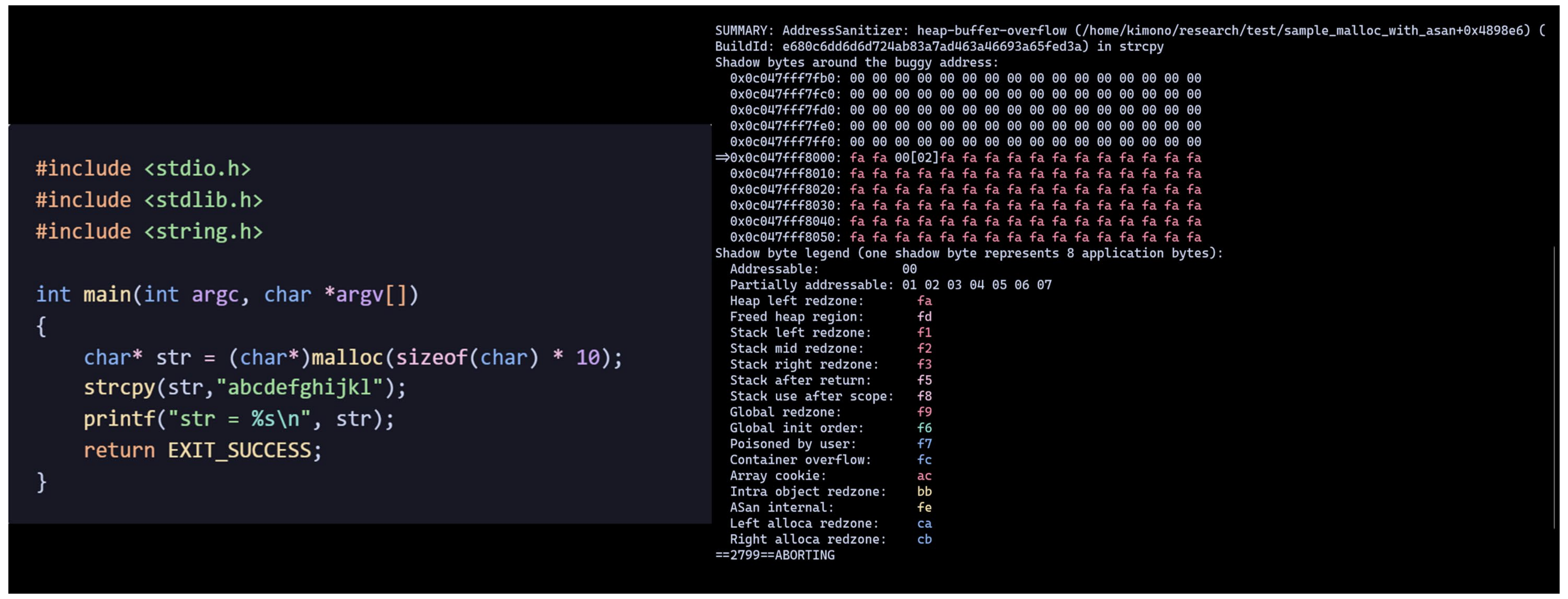
1. We researched on the inner workings of ASAN and identified the important components.
2. The backbone of ASAN involves shadow memory, which stores metadata about the program's memory usage.
3. Shadow memory indicates whether memory is accessible and its state, such as if it is accessible, partially accessible, or inaccessible.
4. The correspondence between program memory and shadow memory is one shadow memory byte for every 8 bytes of program memory.
5. A shadow memory byte of 0 indicates that all 8 bytes are accessible, while other values from 1 to 7 represent partial accessibility, and -1 means no memory is accessible.
6. The ASAN Shadow Memory address calculation formula allows the computation of shadow memory addresses.
7. Instrumentation involves checking the shadow memory byte before memory access, introducing a time overhead for enhanced security.
8. The runtime library in ASAN manages shadow memory, replacing standard memory allocation functions with specialized implementations.
9. Malloc allocates redzones around memory regions to detect overflows and underflows, with redzone size influencing detection scope.
10. Free function poisons memory regions and places them in quarantine as a FIFO queue.
11. Redzones for global objects are generated at compile time.



```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int stack_arr[5];
    stack_arr[6] = 2; // accessing out-of-bound memory
    return 0;
}
```

Stack Redzones

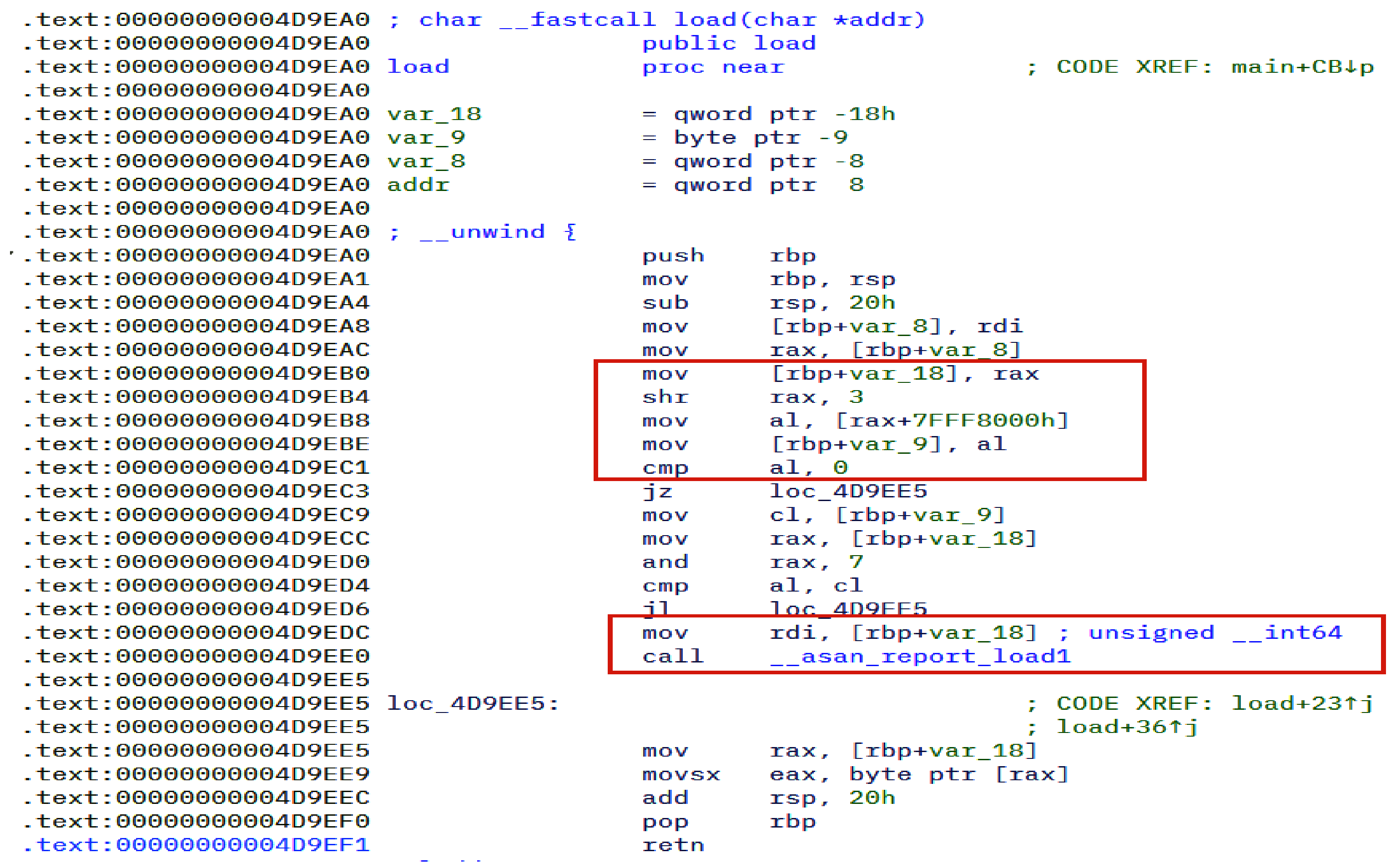


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char* str = (char*)malloc(sizeof(char) * 10);
    strcpy(str,"abcdefghijkl");
    printf("str = %s\n", str);
    return EXIT_SUCCESS;
}
```

```
SUMMARY: AddressSanitizer: heap-buffer-overflow (/home/kimono/research/test/sample_malloc_with_asan+0x4898e6) (
BuildId: e680c6dd6d6d724ab83a7ad463a46693a65fed3a) in strcpy
Shadow bytes around the buggy address:
  0x0c047fff7fb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c047fff7ff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c047fff8000: fa fa 00[02]fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff8010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff8020: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff8030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff8040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c047fff8050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==2799==ABORTING
```

Heap overflow when ASAN is enabled



```
.text:00000000004D9EA0 ; char __fastcall load(char *addr)
.text:00000000004D9EA0                 public load
.text:00000000004D9EA0 load            proc near               ; CODE XREF: main+CB↓p
.text:00000000004D9EA0
.text:00000000004D9EA0 var_18          = qword ptr -18h
.text:00000000004D9EA0 var_9           = byte ptr -9
.text:00000000004D9EA0 var_8           = qword ptr -8
.text:00000000004D9EA0 addr            = qword ptr  8
.text:00000000004D9EA0
.text:00000000004D9EA0 ; __unwind {
.text:00000000004D9EA0                 push    rbp
.text:00000000004D9EA1                 mov     rbp, rsp
.text:00000000004D9EA4                 sub     rsp, 20h
.text:00000000004D9EA8                 mov     [rbp+var_8], rdi
.text:00000000004D9EAC                 mov     rax, [rbp+var_8]
.text:00000000004D9EB0                 mov     [rbp+var_18], rax
.text:00000000004D9EB4                 shr     rax, 3
.text:00000000004D9EB8                 mov     al, [rax+7FFF8000h]
.text:00000000004D9EBE                 mov     [rbp+var_9], al
.text:00000000004D9EC1                 cmp     al, 0
.text:00000000004D9EC3                 jz      loc_4D9EE5
.text:00000000004D9EC9                 mov     cl, [rbp+var_9]
.text:00000000004D9ECC                 mov     rax, [rbp+var_18]
.text:00000000004D9ED0                 and     rax, 7
.text:00000000004D9ED4                 cmp     al, cl
.text:00000000004D9ED6                 jl      loc_4D9EE5
.text:00000000004D9EDC                 mov     rdi, [rbp+var_18] ; unsigned __int64
.text:00000000004D9EE0                 call    __asan_report_load1
.text:00000000004D9EE5
.text:00000000004D9EE5 loc_4D9EE5:                             ; CODE XREF: load+23↑j
.text:00000000004D9EE5                                         ; load+36↑j
.text:00000000004D9EE5                 mov     rax, [rbp+var_18]
.text:00000000004D9EE9                 movsx   eax, byte ptr [rax]
.text:00000000004D9EEC                 add     rsp, 20h
.text:00000000004D9EF0                 pop     rbp
.text:00000000004D9EF1                 retn
```

Checks Added by instrumentation module

Problems

1. ASAN is not very well documented. Current documentation is very concise, not very user friendly and lacks detailed explanation. On top of it, resources for learning ASAN is very scattered. The error messages produced by ASAN are difficult to interpret.
2. ASAN has high memory and time overhead. Average slowdown caused by ASAN is 73% [1].

Future Work

Currently, we are conducting tests on various programs, calculating the overall memory and time overhead caused by ASAN. We are in the process of identifying the components that are causing the overhead, and then we plan to enhance the performance of these components by improving their functionality.

Background Study

1. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in 2012 USENIX Annual Technical Conference (USENIX ATC 12), Jun. 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany
2. Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating Address Sanitizer," in 31st USENIX Security Symposium (USENIX Security 22), Aug. 2022, pp. 4345–4363. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-yuchen
3. https://github.com/gcc-mirror/gcc/blob/releases/gcc-12.2.0/gcc/asan.cc