Sanitize for your Sanity: Sanitizers tools for Modern C++

Bloomberg

Engineering

Meeting C++ 2025
November 7, 2025

Evgenii Seliverstov
Senior Software Engineer

TechAtBloomberg.com

Intro

I am Evgenii Seliverstov

Senior Software Engineer in Bloomberg

PhD and academic research on GPUs and compilers

Write mostly C++ and Rust

Passionate about system and memory safety

Opinions expressed in this talk are solely my own and do not express the views or opinions of my employer

Slides

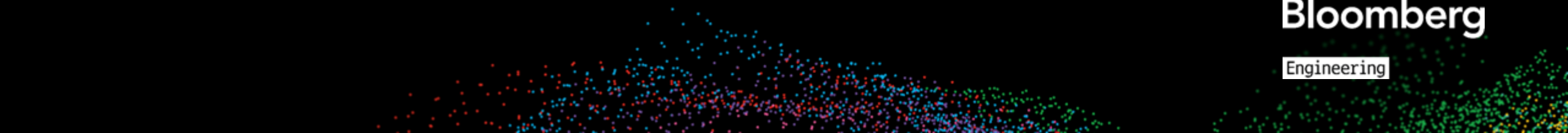
@theirix

Bloomberg

Engineering

Outline

- Approaches
- Build
- Sanitize
- Case study
- Use
- Stay sane

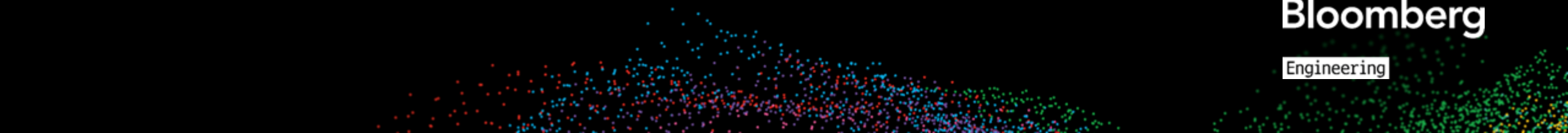


Bloomberg

Engineering

Approaches to ensure safety

- Static analysis — no source code or binary changes
- Runtime analysis via emulation — no source code or binary changes
- Instrumentation — requires binary changes
- Rewrite your code — requires source code changes



Bloomberg

Engineering

Approaches to ensure safety

- Static analysis — no source code or binary changes
  - clang-tidy
  - cppcheck
  - SonarQube
  - PVS-Studio
  - CodeQL
- Runtime analysis via emulation — no source code or binary changes
  - valgrind
  - helgrind
- Instrumentation — requires binary changes
  - LLVM sanitizers
  - Memory tagging
  - dmalloc
  - IBM Purify
- Rewrite your code — requires source code changes

Bloomberg
Engineering

What are sanitizers?

Major sanitizer tools

- ✅ Address sanitizer
- ⚠️ Memory sanitizer
- ✅ Undefined sanitizer
- ⚠️ Thread sanitizer

Extra sanitizer tools

- 🚧 Type sanitizer
- 🚧 Data flow
- ❓ Control flow
- ❓ Safe stack
- ❓ Real-time sanitizer

Bloomberg

Engineering

Build

- How to build?
- How to package?
- Which flags to use?

Bloomberg

Engineering

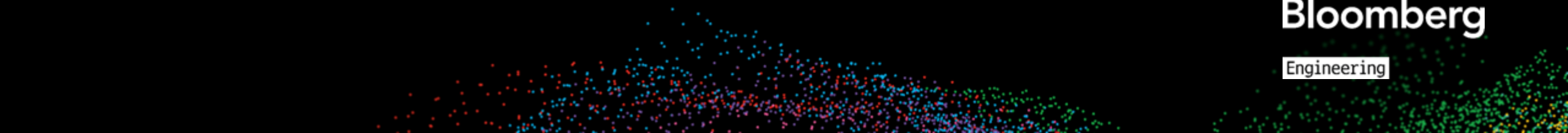
Build system integration: Example

Easy when it's a simple app

One directive only

Is that all?

```
# App
cmake_minimum_required(VERSION 3.14)
project(main)
add_executable(main main.cxx)

target_compile_options(main PUBLIC -g -O1 -fsanitize=address)
```



Bloomberg

Engineering

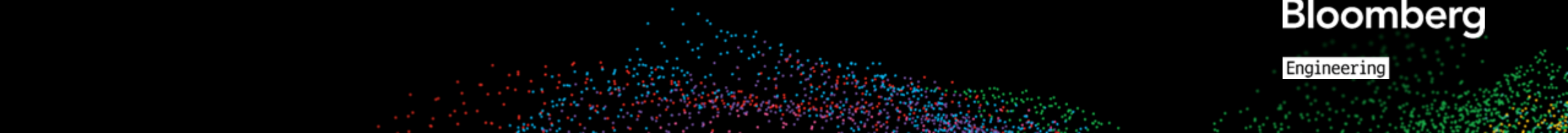
Build system integration: Library example

Static libraries: app → libfoo

Linker must be instructed about -fsanitize=address too

```
# App
cmake_minimum_required(VERSION 3.14)
project(main)
add_executable(main main.cxx)

target_compile_options(main PUBLIC
      -g -fsanitize=address)
target_link_libraries(main PUBLIC
        -fsanitize=address)
```

```
# Library
cmake_minimum_required(VERSION 3.14)
project(foo)
add_library(foo STATIC foo.cpp)

target_compile_options(foo PUBLIC
     -g -fsanitize=address)
```



Bloomberg

Engineering

Build system integration: Ways to enable

Extremely flexible

All options require a custom CMake invocation:

1. In-source changes

```
target_compile_options(foo …)
```

2. CMake option for CXX flags:

```
-DCMAKE_CXX_FLAGS=-fsanitize=address -fno-omit-frame-pointer -fno-common
```

3. CMake option:

```
-DENABLE_ASAN=True
```

4. Custom CMake build type:

```
-DCMAKE_BUILD_TYPE=ReleaseAsan
```

Bloomberg

Engineering

Build system integration: CMake toolchain

Just a file with `set` directives

No changes for the application

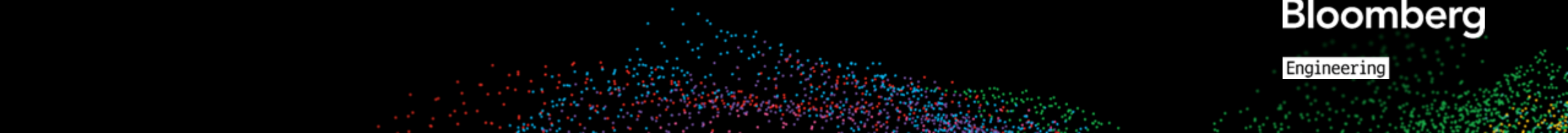
Toolchain file isolates all build flags

Standardized way

Flag -DCMAKE_TOOLCHAIN_FILE=./toolchain.cmake

Via CMake presets — IDE/CI friendly

```
# ~/toolchain.cmake

set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_COMPILER clang++)
```



Bloomberg

Engineering

Package manager integration: Conan

Example for open source JFrog Conan package manager

Conan via profiles

Enable global flags per-profile

Not part of a package

Need to rebuild all projects locally

Profile — Conan-specific system manifest about compilers, CMake flags and paths

```
# ~/.conan2/profiles/asan
include(default)

[env]
CC=/opt/clang/bin/clang
CXX=/opt/clang/bin/clang++
CFLAGS=-fsanitize=address -fno-omit-frame-pointer
LDFLAGS=-fsanitize=address
```

Package manager integration: Conan

Conan supports CMake toolchains natively

Same profile

Standard toolchain is auto-generated for each build: build/conan_toolchain.cmake

Recommended way of extending CMake: Conan toolchain via user toolchain

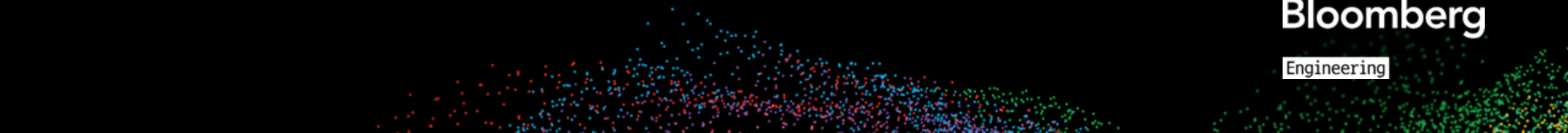
```
# ~/.conan2/profiles/asan
include(default)

[conf]
tools.cmake.cmaketoolchain:user_toolchain+=
{{profile_dir}}/asan.add.toolchain.cmake
```

```
# ~/.conan2/profiles/asan.add.toolchain.cmake

set(CONAN_CXX_FLAGS "-fsanitize=address
                     -fno-omit-frame-pointer")

add_compile_options("-fsanitize=address")
link_libraries("-fsanitize=address")
```

Compiler flags

- Do not optimize call frames
- Use debug info → Readable stack frames
- Handle uninitialized variables

- Do not turn off optimisation
- Runs with adequate speed
- Avoid behavioural changes → Equivalent behaviour
- More about side-effects later



Bloomberg

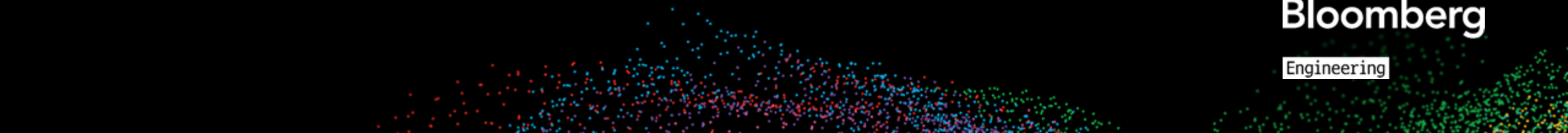
Engineering

Build

With modern C++ build practices:

- No source code changes
- No build flags changes
- No build system manifest changes

Looks impressive!



Bloomberg

Engineering

Sanitize

What's inside?

How does it work?

Dive deep into sanitizers

- Address
- Memory
- Thread
- Leak

Explore typical problems

Bloomberg

Engineering

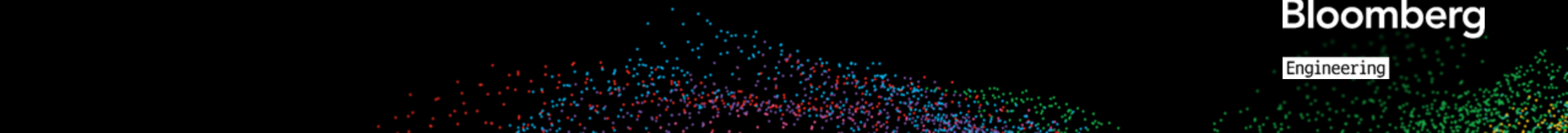
Address sanitizers

The most used and useful one

Supported in Clang and GCC

Classes of errors:

- out-of-bounds
- use-after-free
- double-free
- use-after-return



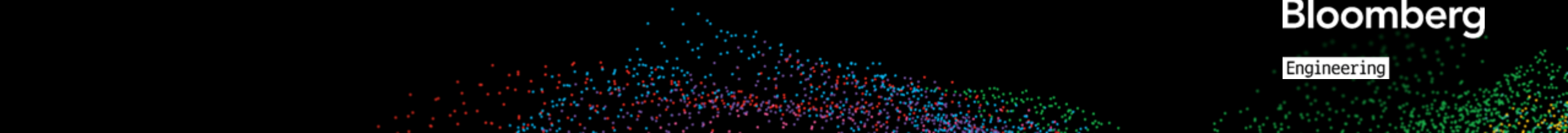
Bloomberg

Engineering

How do address sanitizers work?

- During compile-time:
  - Instrument program
  - Replace memory access with special calls
  - On memory access, check if memory is accessible
  - For stack memory, guard it with special buffers

- During load-time:
  - Intercept memory-related libc calls with its own
  - Provide rich diagnostics when a program crashes



Bloomberg

Engineering

Let’s finally crash something

Heap break.

Array at 0x6020000000c0 — definitely heap (high addresses)

Bad access is 0x6020000000c8 — index 6 after 16-byte array

```
==53423==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x6020000000c8 a
t pc 0x0001009bc35d bp 0x7ff7bf545ef0 sp 0x7ff7bf545ee8
READ of size 4 at 0x6020000000c8 thread T0
    #0 0x1009bc35c in main3() main.cpp:30
    #1 0x1009bc418 in main main.cpp:34
    #2 0x10f58352d in start+0x1cd (dyld:x86_64+0x552d)

0x6020000000c8 is located 8 bytes to the right of 16-byte region [0x6020000000b0,0
x6020000000c0)
allocated by thread T0 here:
    #0 0x100e5f31d in wrap__Znam+0x7d (libclang_rt.asan_osx_dynamic.dylib:x86_64h+
0x5c31d)
    #1 0x1009bc18b in main3() main.cpp:22
    #2 0x1009bc418 in main main.cpp:34
    #3 0x10f58352d in start+0x1cd (dyld:x86_64+0x552d)
```

```
void main3() {
    int* param =
                new int[4];
    param[0] = 1;
    param[1] = 2;
    param[2] = 3;
    param[3] = 4;
    int val = param[6];
}
```

clang 14
x86_64
macOS

Bloomberg
Engineering

How does it work: Shadow memory

Address 0x1c0400000010 in the map

But faulty memory is 0x6020000000c8

Shadow memory region

Mac: 0x100000000000..0x1fffffffffff

Linux: 0x00007fff8000..0x10007fff7fff

Real memory is tracked in shadow

- White bytes — addressable
- Colored bytes — poisoned

On poisoned access, raise the error immediately.

```
SUMMARY: AddressSanitizer: heap-buffer-overflow main.cpp:30 in main3()
Shadow bytes around the buggy address:
  0x1c03fffffff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1c03fffffff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1c03fffffff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1c03fffffff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow memory fa fd fd fa fa 00 00 fa fa 00 04 fa fa 00 fa
=>0x1c0400000010 fa fa 00 04 fa fa 00 00 fa[fa]fa fa fa fa fa fa
  0x1c0400000020: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0400000030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0400000040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0400000050: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x1c0400000060: 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
```

Real memory
0x6020000000c8

Bloomberg
Engineering

How does it work: Accessing

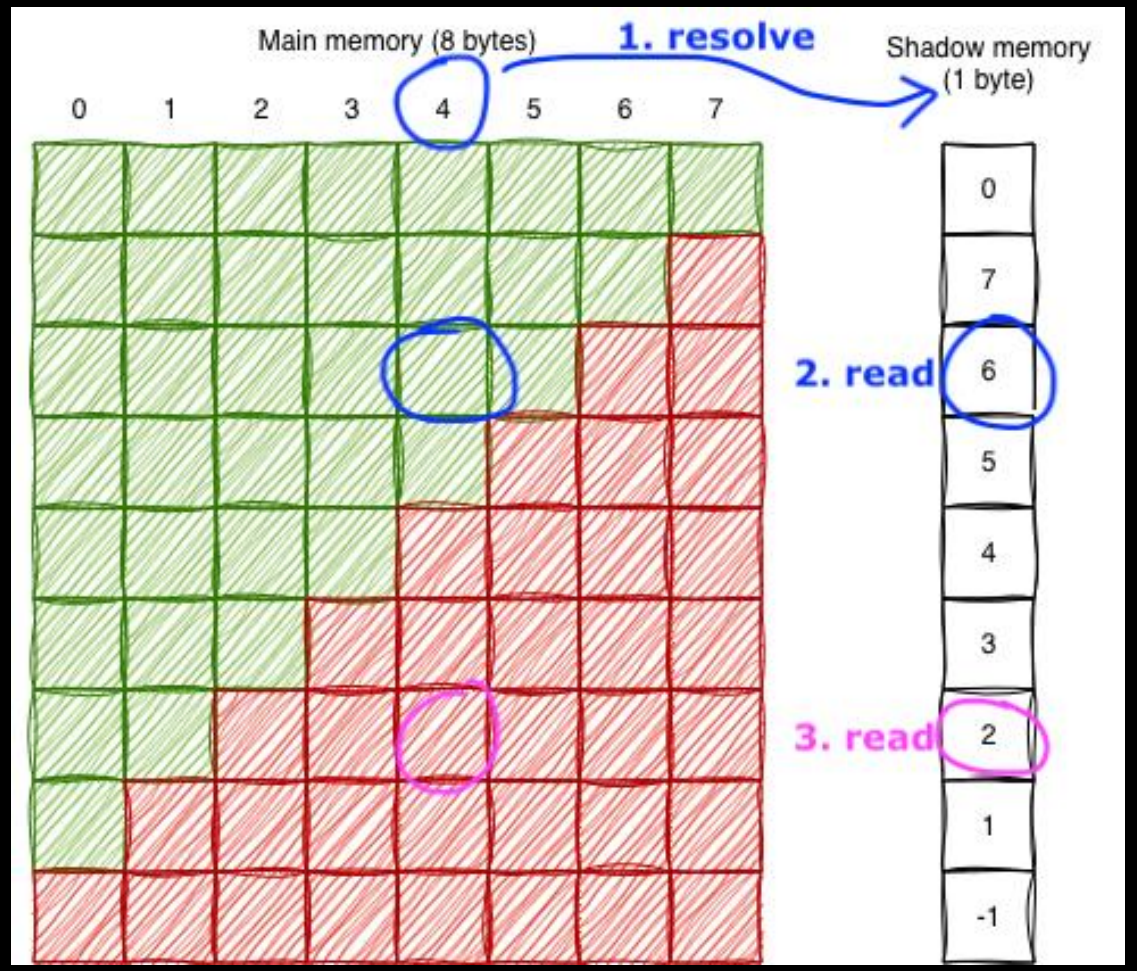
Twice memory usage in shadow?

Optimisation:

- Each 8-byte segment is coded in 1 shadow byte
- Heap allocations are aligned to 8 bytes

Accessing a byte in main memory is **instrumented** via _asan_report_load4 functions:

1. Resolve a shadow address
2. If shadow is unpoisoned → ok (scenario 1-2)
3. If shadow is fully poisoned → report right away
4. If shadow is partially poisoned → check (scenario 1-3) if a byte lies in a green part



Bloomberg
Engineering

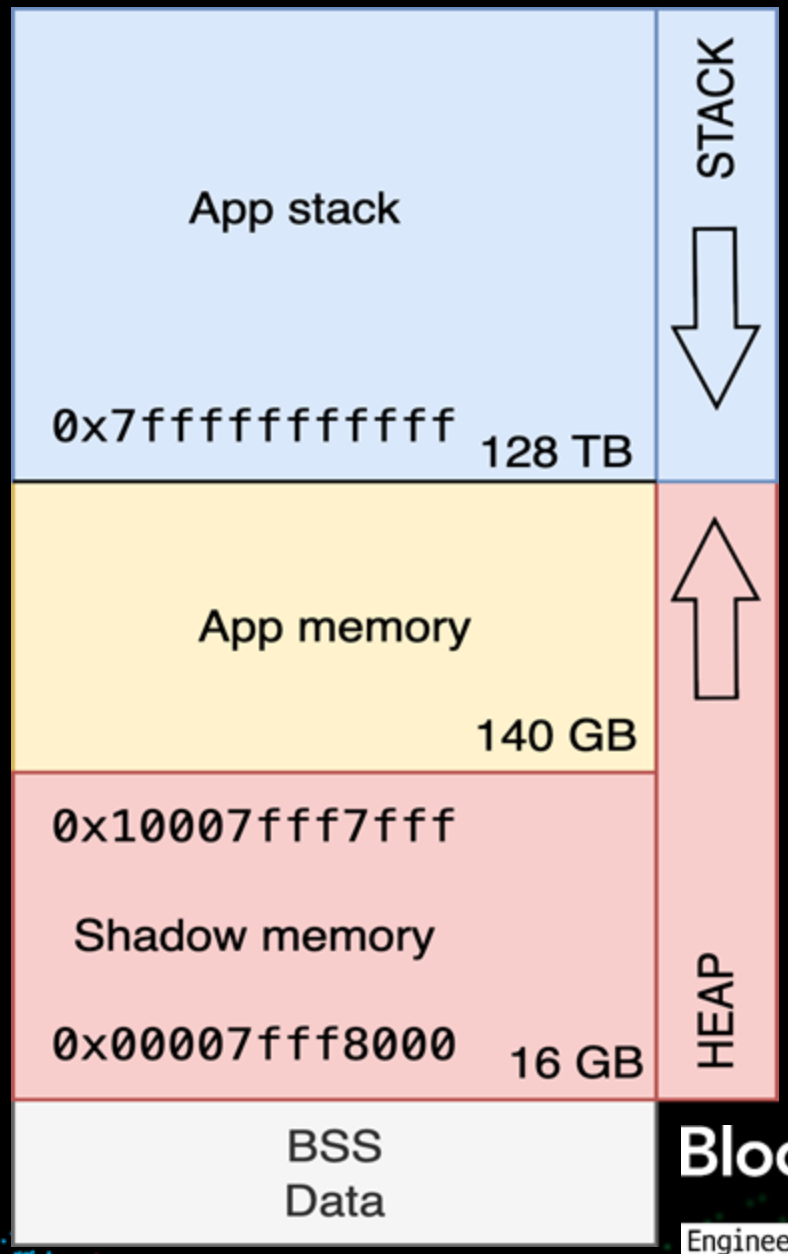
Two kinds of memory

- Stack
- Heap

Stack memory is different

Runtime cannot control the address

Address sanitizer uses another approach



Linux
64-bit

Bloomberg
Engineering

Let’s crash the stack now

Each protected stack area is surrounded with red zones

Red zones are mapped to shadow memory

Aligned to 32 bytes

```
void main2()
{
    int param[5] = {1, 2, 3, 4, 5};
    char str[] = "testingtests";
    int val = param[6];
}
```

20 bytes for param

12 + 1 bytes for str

```
==92170==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ff7b47c6f38
 at pc 0x00010b73af80 bp 0x7ff7b47c6ef0 sp 0x7ff7b47c6ee8
READ of size 4 at 0x7ff7b47c6f38 thread T0
    #0 0x10b73af7f in main2() main.cpp:16
    #1 0x10b73b3f8 in main main.cpp:34
    #2 0x11144952d in start+0x1cd (dyld:x86_64+0x552d)

Address 0x7ff7b47c6f38 is located in stack of thread T0 at offset 56 in frame
    #0 0x10b73ae2f in main2() main.cpp:12

  This frame has 2 object(s):
    [32, 52) 'param' (line 13) <== Memory access at offset 56 overflows this vari
able
    [96, 109) 'str' (line 14)
```

Bloomberg

Engineering

Shadow memory: Stack allocations

The floor is red lava!

Left red zone — 20 bytes

- 8 bytes - 00
- 8 bytes - 00
- 4 bytes - 04

Mid-red zone — 13 bytes

- 8 bytes - 00
- 5 bytes - 05

```
SUMMARY: AddressSanitizer: stack-buffer-overflow main.cpp:16 in main2()
Shadow bytes around the buggy address:
  0x1ffef68f8d90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1ffef68f8da0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1ffef68f8db0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1ffef68f8dc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1ffef68f8dd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x1ffef68f8de0: f1 f1 f1 f1 00 00 04[f2]f2 f2 f2 f2 00 05 f3 f3
  0x1ffef68f8df0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1ffef68f8e00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1ffef68f8e10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1ffef68f8e20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x1ffef68f8e30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
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
```

Hitting poisoned zone triggers a sanitizer report

Sanitizer knows which stack variable is touched based on reverse mapping

How does it work: Marking memory

When to mark shadow memory

Intercepted functions:

- new/malloc/free/delete
- string functions

Intercept malloc:

- Resolve shadow memory address
- Mark shadow memory as addressable (0x00 .. 0x07)

Intercept free:

- Resolve shadow memory address
- Mark as freed (0xfd)

Heap

Allocation (malloc, new)
Free (free, delete)

Compiler and function support

Stack

Frame enter
Frame exit

Only compiler support

Bloomberg

Engineering

Address sanitizer: Costs

A matter of switching compiler flags

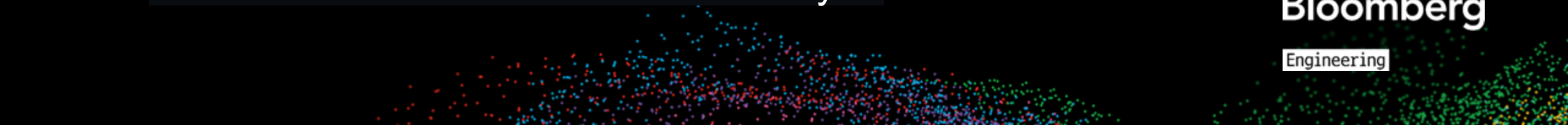
Typical slowdown: 2x

Binary size overhead: 2-3x

Memory overhead: 2-3x (mostly for stack)

Optimisations are applied everywhere:

- 8-to-1 shadow memory
- More compact shadow memory layout
- Merging checks for subsequent memory access
- Avoid extra checks based on flow analysis



Bloomberg

Engineering

Make it lighter: 1/3

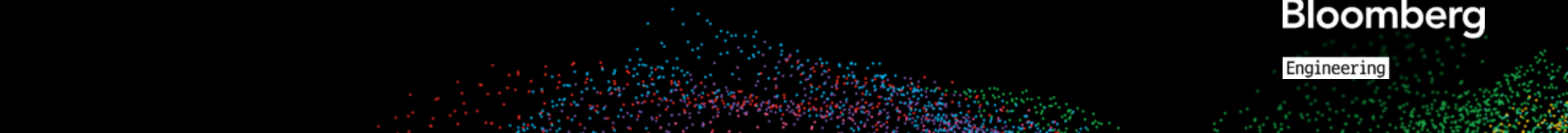
Compile-time or run-time

Skip instrumentation:

- Critical for performance
- Different correctness requirements
- Disable for function — attribute `__attribute__((no_sanitize("address"))`
- Disable instrumenting a function or a type `-fsanitize-ignorelist=ignores.txt`

Run-time compatibility:

- Disable checks in external non-instrumented code
- `ASAN_OPTIONS=suppressions=ignores.txt`



Bloomberg

Engineering

Make it lighter: 2/3

Tune down instrumentation

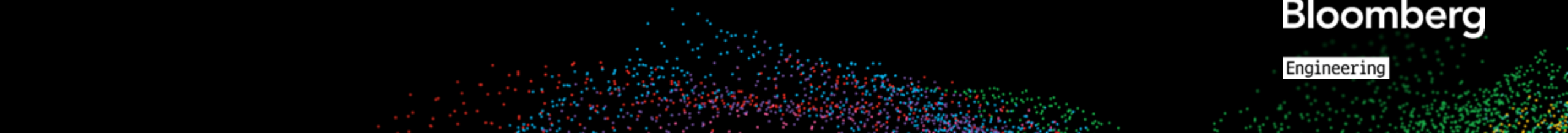
- Leaner code
- Disable use-after-return -fsanitize-address-use-after-return=never
- Disable use-after-scope -fsanitize-address-use-after-scope
- Disable use-after-return runtime check
  ASAN_OPTIONS=detect_stack_use_after_return=0.

Disable leak tracking

- ASAN_OPTIONS=detect_leaks=0

Trap mode

- Raise a signal SIGTRAP for program to handle -fsanitize-trap=all



Bloomberg

Engineering

Make it lighter: 3/3

Experimental hardware acceleration

Different flavour -fsanitize=hwaddress

Tagged memory support

AArch64 is supported

- mobile devices and mac CPUs
- server ARM CPUs

Intel is ongoing:

- Clang 14 (2022) / gcc 13 (2025)
- Intel LAM (Linear Address Masking) — Arrow Lake (2025)

Memory Tagging and how it improves C/C++ memory safety
arXiv:1802.09517

GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production
arXiv:2311.09394

Bloomberg
Engineering

Leak Sanitizer

A part of address sanitizer

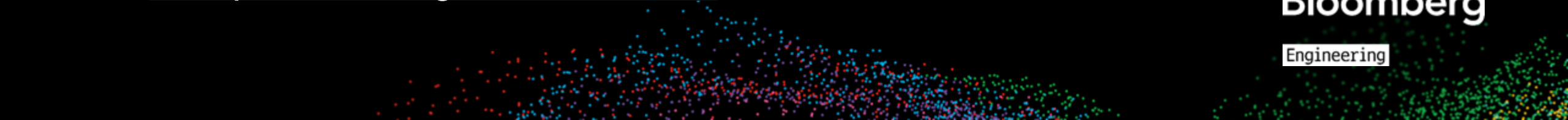
Enabled for free:

- Address sanitizer already intercepts malloc / free
- It’s enough to track memory

Reports leaks on program exit

Performance considerations:

- Leak check phase is expensive (pause threads, flood fill pointers)
- Use leak sanitizer without address sanitizer (better performance before leak check)
- Compile-time flag -fsanitize=leak



Bloomberg

Engineering

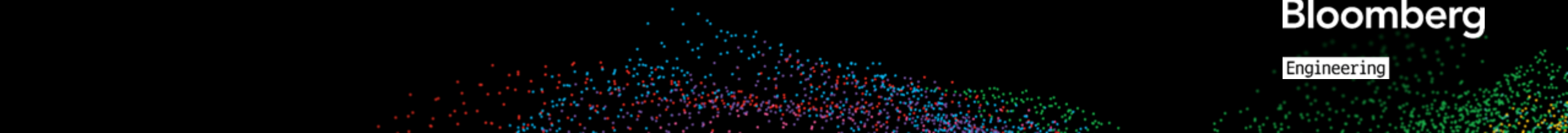
Leak Sanitizer: Problems

Opinionated checks

Enabled depending on platform — ASAN_OPTIONS=detect_leaks=1

Specific program design

- Allocate a lot of things in main()
- Don’t care about deallocating on exit — not a leak
- Typically, a loop — incoming HTTP/RPC requests, Apache Kafka messages
- Ideally measure leaks for a processing iteration



Bloomberg

Engineering

Leak Sanitizer: Fine-grained checks

Tailored leak measurement

Internal LLVM function
__lsan_do_recoverable_leak_check

Accumulated leaks, not differential

```cpp
extern "C" { int __lsan_do_recoverable_leak_check(); }

void main4()
{
    for (int counter = 0; counter < 4; ++counter)
    {
        process(counter);
        if (__lsan_do_recoverable_leak_check())
        {
            std::cout << "Leaks detected\n";
        }
        std::this_thread::sleep_for(
            std::chrono::seconds(5));
    }
}
```

```
==3977877==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 1 byte(s) in 1 object(s) allocated from:


==3977877==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 3 byte(s) in 2 object(s) allocated from:
```

Bloomberg
Engineering

Beyond Leak Sanitizer: jemalloc

Drop-in runtime replacement for malloc

Automatic mode:

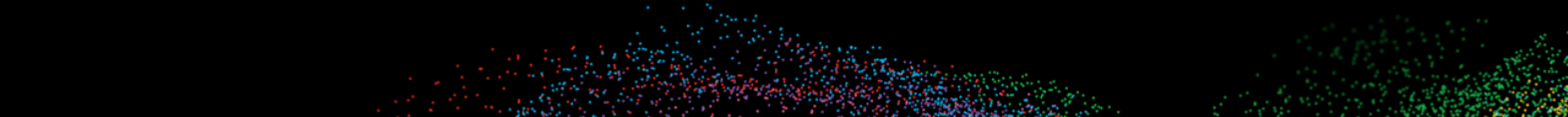
- On exit, similar to leak-sanitizer: export MALLOC_CONF="prof:true”

Manual continuous mode:

- Time- or size-based report: export MALLOC_CONF="prof:true,lg_prof_interval:5”

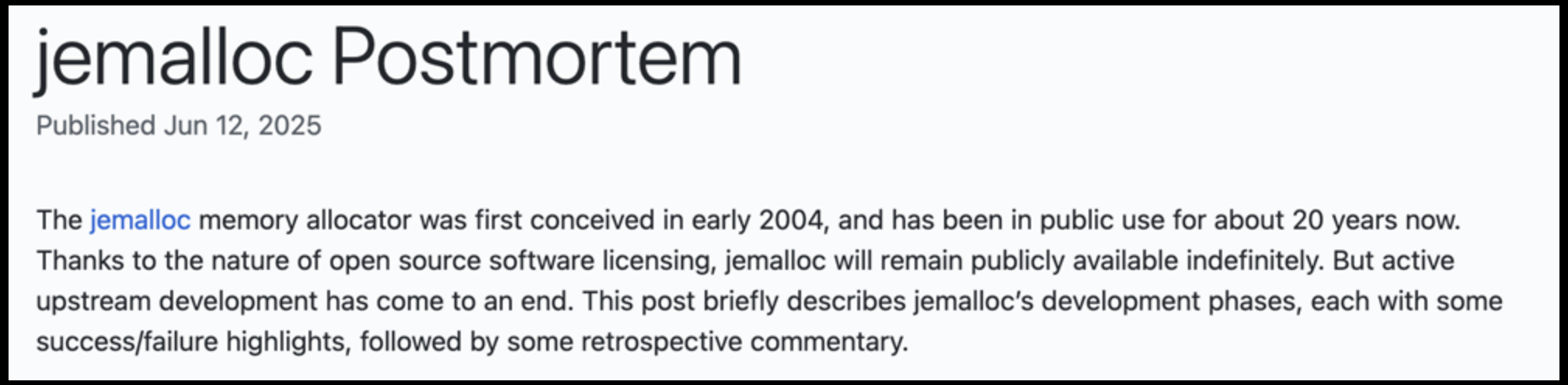
Manual differential mode:

- Compile and link with jemalloc support: #include <jemalloc/jemalloc.h>
- Invoke mallctl("prof.active", …) to start leak tracking
- Invoke mallctl("prof.dump", …) to compare and report a difference
- Check profile file with a jeprof utility



Beyond Leak Sanitizer

jemalloc post-mortem



jemalloc Postmortem

Published Jun 12, 2025

The jemalloc memory allocator was first conceived in early 2004, and has been in public use for about 20 years now. Thanks to the nature of open source software licensing, jemalloc will remain publicly available indefinitely. But active upstream development has come to an end. This post briefly describes jemalloc's development phases, each with some success/failure highlights, followed by some retrospective commentary.

Google's tcmalloc

More or less equivalent

Both not compatible with sanitizers

Bloomberg

Engineering

Platform support

Address sanitizer

- ✅ Linux, macOS on x86 / x86_64

Leak sanitizer

- ✅ clang on Linux x86_64
- ❌ Apple clang on macOS x86_64
- ✅ LLVM clang on macOS x86_64
- ❌ macOS AArch64

Thread sanitizer

- ✅ Linux, macOS on x86_64

Undefined behaviour sanitizer

- ✅ Everywhere

🤔 Windows — MSVC, clang/MSVC, mingw?

Bloomberg

Engineering

Address sanitizer: False positives

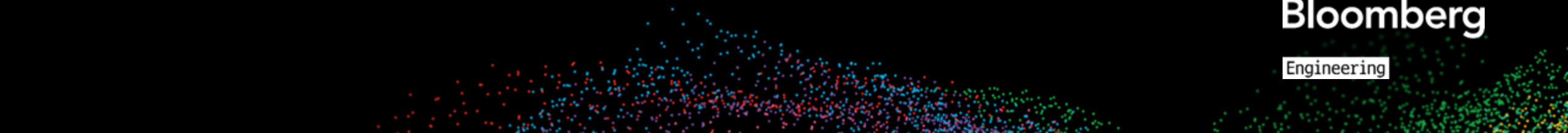
Very rarely

Mixing instrumented and non-instrumented code

Totally fine with address sanitizer (NOT thread sanitizer)

Typical example: container overflow

- Source code of a container is not instrumented — it belongs to libc++ library
- Inline code is instrumented — it is compiled in application
- Mitigation from app-side: ASAN_OPTIONS=detect_container_overflow=0
- Mitigation from compiler: __sanitizer_annotate_contiguous_container annotation on std containers



Bloomberg

Engineering

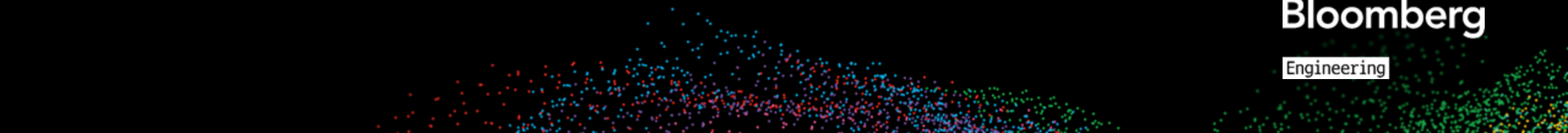
Address sanitizer: False negatives

False negatives for unaligned access

- Unaligned access is bad, however

Hardened code is not sanitised

- Debian and Gentoo hardening with `-D_FORTIFY_SOURCE`



Bloomberg

Engineering

Sanitizers: PIC and PIE

Source of confusion

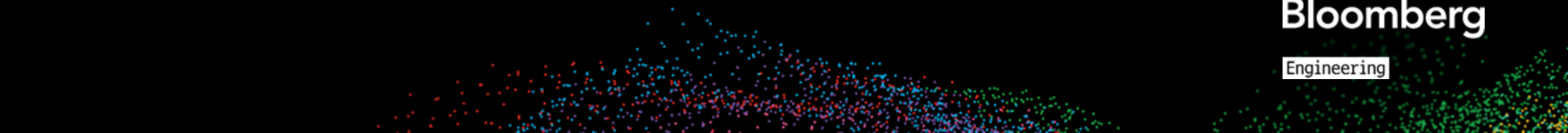
Only thread and memory sanitizers require PIC/PIE

Symptoms during compile-time:

```
g++: error: -fsanitize=thread linking must be done with -pie or –shared

ld: relocation R_X86_64_32S cannot be used when making a shared object;
recompile with -fPIC
```

Symptoms during run-time:

```
==70588==ReserveShadowMemoryRange failed while trying to map
0xdfff0001000 bytes. Perhaps you're using ulimit -v
```



Bloomberg

Engineering

Sanitizers: PIC and PIE

PIC — for shared libraries to be loaded at arbitrary address

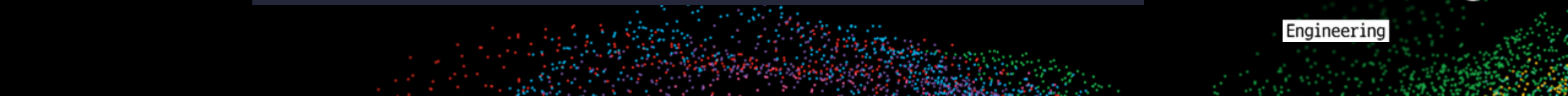
PIE — for executables to support ASLR (load at random address)

Code is built and linked as relocatable via offsets

For shared libraries, set -fPIC compilation flag (not -fPIE)

For executables, set -fPIC or -fPIE compilation flag and -pie linker flag

```
% readelf -h app-pie
app-pie: DYN (Position-Independent Executable file)

% readelf -h app-nonpie
app-nonpie: EXEC (Executable file)
```



Bloomberg

Engineering

Sanitizers: common problems

Memory limits are imposed via PAM (reserve terabytes of virtual, memory)

SELinux with mmap restrictions (used for shadow)

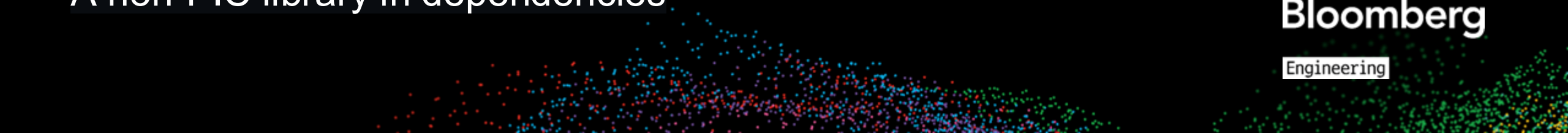
Memory overcommit is disabled (vm.overcommit_memory must be 0 or 1)

Another allocator (tcmalloc, jemalloc)

ASLR is disabled (check kernel.randomize_va_space)

Non-PIE binary

A non-PIC library in dependencies



Bloomberg

Engineering

Sanitizer runtime

Where do new symbols come from?

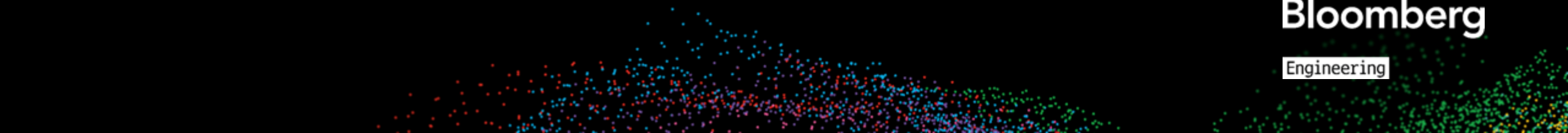
Runtime provides

- Memory load/store functions
- Reporting functions
- Interceptors

LLVM and GCC library implementations

Mix-and-match is prohibited

```
% nm libfoo.a  | grep san
    U __asan_handle_no_return
    U __asan_init
    U __asan_option_detect_stack_use_after_return
    U __asan_register_globals
    U __asan_report_load1
    U __asan_report_load4
    U __asan_report_load8
    U __asan_report_store1
    U __asan_report_store4
    U __asan_report_store8
    U __asan_stack_malloc_0
    U __asan_stack_malloc_2
    U __asan_stack_malloc_3
    U __asan_unregister_globals
    U __asan_version_mismatch_check_v8
```



Bloomberg

Engineering

Sanitizer runtime per platform

Linux Clang — static by default

Linux GCC — shared by default

```
% ldd ./build/app/app | grep san
        libasan.so.8 => /lib/x86_64-linux-gnu/libasan.so.8 (0x00007fa178400000)
```

To link sanitizer statically, specify linker flag `-static-libasan`

macOS — only shared sanitizer

Another mechanism rather than `LD_PRELOAD`

A list of interceptors in `__DATA` section of a binary

Bloomberg

Engineering

Case of Bloomberg

Linux, GCC

Use of CMake toolchains

Statically linked sanitiser -static-libasan

Huge support from tooling and CI

- Different CMake toolchains for prod and instrumented builds
- Build internals are isolated
- -DCMAKE_TOOLCHAIN_FILE=/opt/BBInstrumentationToolchain64.cmake
- -DBB_INSTRUMENTATION_TYPES='ASAN;UBSAN'

Unit tests under sanitisers

Integration tests under sanitisers

Bloomberg

Engineering

Memory Sanitizer

Do not confuse with address sanitizer

Flags: -fsanitize=memory -fsanitize-memory-track-origins=2

Only one class of errors:
Reads from uninitialized memory

```
int* a = new int[10];
a[5] = 0;
// Uninitialized read (MSan)
printf("%d\n", a[1]);
```

```
WARNING: MemorySanitizer:
use-of-uninitialized-value
```

Compare: address sanitizer
Detects illegal reads only

```
int* a = new int[10];
a[5] = 0;
// Illegal read (ASan)
printf("%d\n", a[20]);
```

Bloomberg
Engineering

Memory Sanitizer: Costs

Much more expensive in runtime (2-3x slower)

Stops at first offence

Requires rebuild of all dependencies:

- Practically can run
- False positives when partially instrumented
- Suppressions could help

Requires relocatable binary -fPIE

Not a practical choice

Use ASan instead

Bloomberg

Engineering

Thread Sanitizer

It is hard to make a correct C++ multi-threaded program

Event harder to find bugs

Is it a feasible problem in C++?

Lack of language and library primitives

Fearless Concurrency 🦀 is not achievable

Heuristics to detect races and concurrency problems

CppOnSea 2025
David Rowland
What can C++ learn about thread safety from other languages?

Bloomberg
Engineering

Thread Sanitizer: Heuristics

Best effort

Simple race
Race on a complex object
Notification
Publishing objects without synchronization
Initializing objects without synchronization
Reader Lock during a write
Race on bit field
Double-checked locking
Race during destruction
Data race on vptr
Data race on vptr during construction
Race on free
Race during exit
Race on a mutex
Race on a file descriptor

```
// tsan_report.h

enum ReportType {
  ReportTypeRace,
  ReportTypeVptrRace,
  ReportTypeUseAfterFree,
  ReportTypeVptrUseAfterFree,
  ReportTypeExternalRace,
  ReportTypeThreadLeak,
  ReportTypeMutexDestroyLocked,
  ReportTypeMutexDoubleLock,
  ReportTypeMutexInvalidAccess,
  ReportTypeMutexBadUnlock,
  ReportTypeMutexBadReadLock,
  ReportTypeMutexBadReadUnlock,
  ReportTypeSignalUnsafe,
  ReportTypeErrnoInSignal,
  ReportTypeDeadlock,
  ReportTypeMutexHeldWrongContext
};
```

Bloomberg
Engineering

Thread Sanitizer: Example 1

Let’s race

Simple race condition on a variable

Continues execution on race

```
void main() {
  static int state;
  std::jthread t1([] { state = 44; });
  std::jthread t2([] { state = 2; });
}
```

```
==================
WARNING: ThreadSanitizer: data race (pid=1774829)
  Write of size 4 at 0x560e4b267b2c by thread T2:
    #0 main5()::$_1::operator()() const app/main.cpp:10:30 (app+0xe8638)
  Previous write of size 4 at 0x560e4b267b2c by thread T1:
    #0 main5()::$_0::operator()() const app/main.cpp:9:30 (app+0xe8078)

  Location is global 'main5()::state' of size 4 at 0x560e4b267b2c (app+0x1518b2c)
  Thread T2 (tid=1774832, running) created by main thread at:
    ...
  Thread T1 (tid=1774831, finished) created by main thread at:
    ...
SUMMARY: ThreadSanitizer: data race app/main.cpp:10:30 in main5()::$_1::operator()() const
==================
```

Bloomberg

Engineering

Thread Sanitizer: example 2

Race condition on a variable

Changes execution flow

```
SUMMARY: ThreadSanitizer: data race
          app/main.cpp:27:12
          in
main6()::$_1::operator()() const
```

```cpp
__attribute__((__noinline__)) void blackbox() {}

void main6() {
    bool done = false;

    std::jthread t1([&done] {
      while (!done) blackbox();
      std::cout << "Unreachable" << std::endl;
    });
    std::jthread t2([&done] {
      blackbox();
      done = true;
    });
}
```

Thread Sanitizer: costs

Much more expensive in runtime (10x slower)

Typical slowdown: 10x

Memory overhead: 5-10x (more than ASan)

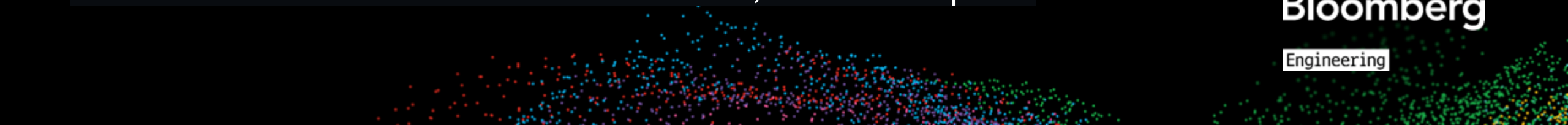
Requires relocatable binary -fPIE

Same approach to exclude code

A lot of false positives

libstdc++ cannot be instrumented — so, it's intercepted!

```
% nm libclang_rt.tsan-x86_64.a | \
  grep ' T ___interceptor_' | wc -l
570

___interceptor_pthread_mutex_lock
___interceptor_pthread_join
___interceptor_popen
```



Bloomberg

Engineering

Case study

We know how they work

How to use them in the real world:

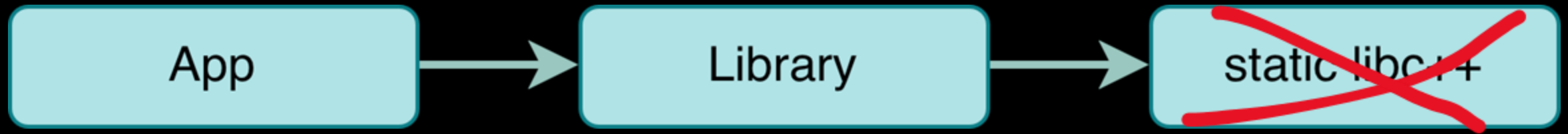
- Cross-boundary interactions
- Static vs. shared libraries
- False positives and negatives

Cross-language support

Bloomberg

Engineering

Sanitizers in complex projects



❌ Static linking is not supported at all

✅ Shared libc++ or libstdc++

libc++ cannot be not instrumented

Use interceptor functions instead (sanitizer runtime)

Bloomberg

Engineering

Address sanitizers in complex projects: Static libraries

Instrument both app and library

✅ Perfect case

Allocations from library are also tracked

Allocations can cross a boundary

No false positives or negatives

libc++ doesn’t need to be instrumented

Bloomberg
Engineering

Address sanitizers in complex projects: Static libraries

Only a library is instrumented, but not an application

Doesn’t work without build changes

```
f.cpp:(.text.asan.module_ctor[asan.module_ctor]+0x5): undefined reference to `__asan_init'
```

Executable must link to sanitizer library

Provide a linker flag `-fsanitize=address`

✅ Fine since you control linking

Bloomberg

Engineering

Address sanitizers in complex projects: Static libraries

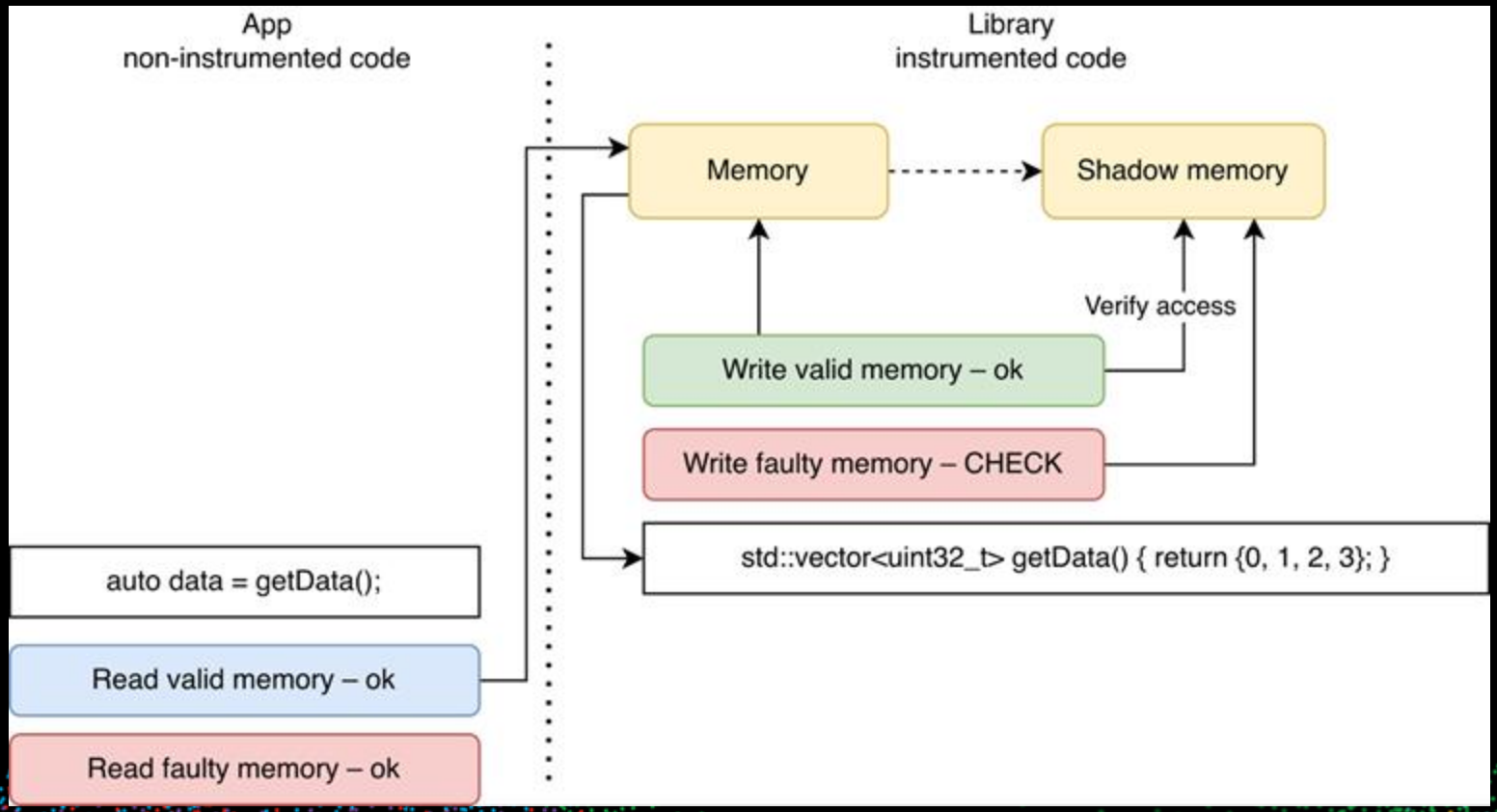
⚠ False negatives

Cross-boundary situation:

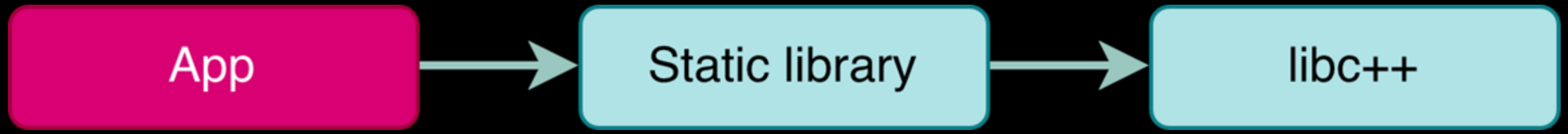
Memory allocated in the library

Make a bad access in the app

It isn’t flagged by the sanitizer



Address sanitizers in complex projects: Static libraries



Only executable is instrumented

Works fine

Allocations from executable are tracked

⚠ Allocations from the library are not tracked

Bloomberg
Engineering

Address sanitizers in complex projects: Dynamic libraries

Instrument both app and library

✅ Perfect case

No false positives

No false negatives

Note: link with shared ASan runtime

Bloomberg

Engineering

Address sanitizers in complex projects: Dynamic libraries

Interesting case — only the library is instrumented

1. Originally executable wasn't aware of sanitizers and linking libasan runtime
2. Rebuild library with ASan
3. Got this:

```
/build/app/app: symbol lookup error: /build/libfoo/libfoo.so:
undefined symbol: __asan_option_detect_stack_use_after_return
```

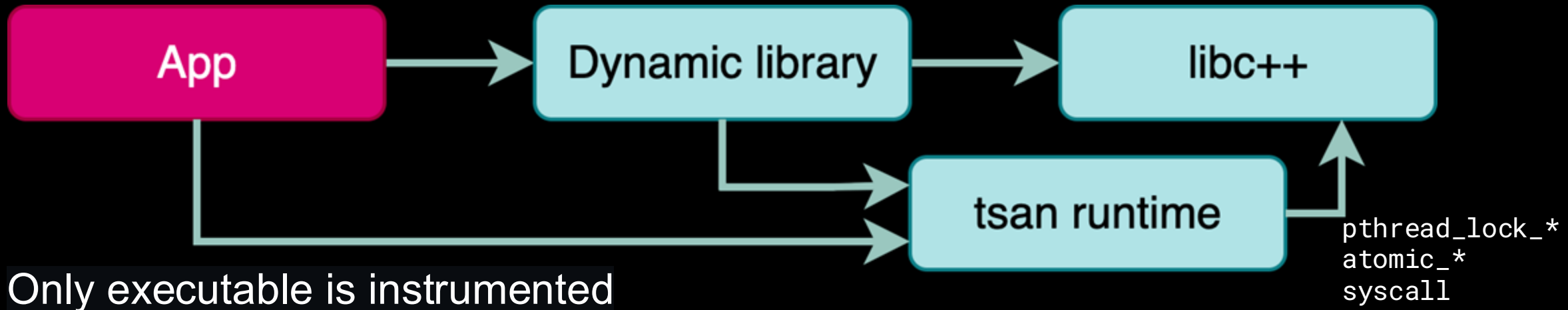
Solution: runtime must be preloaded:

```
LD_PRELOAD=/lib/x86_64-linux-gnu/libasan.so.8 ./app
```

Bloomberg

Engineering

Thread sanitizers in complex projects



Only executable is instrumented

Bad behaviour in library is not checked — expected ⚠️

Good behaviour in a library is checked — no false positives ✅

- Would expect false positive since an app isn't aware
- All pthread_lock_* dynamic calls are intercepted by TSan runtime
- But only if it is intercepted…

Bloomberg
Engineering

Thread sanitizers in complex projects: Example 1

Let's crash with spinlocks!

```cpp
// App (instrumented)
int state = 0;

std::jthread t1([&] {
    acquire_lock(); //TSAN: normal function
    ++state;        //TSAN: read from T1
    release_lock();
});
std::jthread t2([&] {
    acquire_lock(); //TSAN: nothing special
    ++state;        //TSAN: read from T1
    release_lock();
});
```

```cpp
// Shared library (non-instrumented)
static std::atomic<bool> spinlock{false};

void acquire_lock() {
    while (spinlock.exchange(
       true, std::memory_order_acquire)) ;
}

void release_lock() {
    spinlock.store(false,
       std::memory_order_release);
}
```

⚠ False positive (i.e., error is reported)

But TSan could analyse acquire_lock / release_lock if instrumentation enabled

Thread sanitizers in complex projects: Example 2

Let’s crash with system mutex!

```
// App (instrumented)
int state = 0;

std::jthread t1([&] {
    acquire_lock(); //TSAN: known
    ++state;         //TSAN: read T1
    release_lock();
});
std::jthread t2([&] {
    acquire_lock(); //TSAN: known
    ++state;        //TSAN: read T1
    release_lock();
});
```

```
// Shared library (non-instrumented)
static pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;

// pthread_mutex_lock is intercepted
void acquire_lock() { pthread_mutex_lock(&mutex); }

// pthread_mutex_lock is intercepted
void release_lock() { pthread_mutex_unlock(&mutex); }}
```

✅ No false positive (i.e., no errors reported)

Function pthread_mutex_lock is intercepted in runtime

Bloomberg

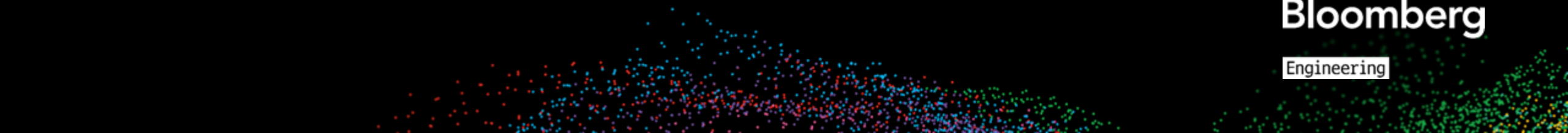
Engineering

Address sanitizers in cross-language projects

Integrating libraries in different languages

How does it work on cross-language barriers?

Deal with RIIR 🦀



Bloomberg

Engineering

Address sanitizers in Rust/C++: pure Rust app

Rust can also crash

In unsafe

So could an unsafe Rust dependency

Sanitizers can help

```
// RUSTFLAGS="-Z sanitizer=address" cargo run -v
fn main() {
    let mut v = vec![0, 1, 2, 3];
    let mut ptr: *mut i32 = &mut v[0] as *mut i32;
    ptr = ptr.wrapping_add(10);
    let ub: i32 = unsafe { *ptr };
}
```

Bloomberg

Engineering

Address sanitizers in cross-language projects

Dynamic C++ library — instrumented

Client application written in Rust — instrumented

Autogenerated bridge — instrumented

Must build Rust app with sanitizers

All components use static runtime

Bloomberg

Engineering

Address sanitizers in Rust/C++: bridge

Shared C++ library libfoo.so is linked to static ASan runtime

Using cxx crate to link to C++ library foo

```
#[cxx::bridge]
mod ffi {
  unsafe extern "C++" {
      // int processString(const std::string& param, size_t len);
      include!("/usr/local/include/foo.h");
      fn processString(param: &CxxString, len: usize) -> i32;
} }
```

Autogenerated bridge with glue types

Static library cxxbridge.a

Bloomberg

Engineering

Address sanitizers in Rust/C++: bridge

Call C++ function from Rust code

Pass memory address from stack

Autogenerated bridge with glue types

Static library cxxbridge.a

```
#[cxx::bridge]
mod ffi {
  unsafe extern "C++" {
      // int processString(const std::string& param, size_t len);
      include!("/usr/local/include/foo.h");
      fn processString(param: &CxxString, len: usize) -> i32;
} }

cxx::let_cxx_string!(param = "testingtests");
let result = ffi::processString(&param, 100);
```

Bloomberg Engineering

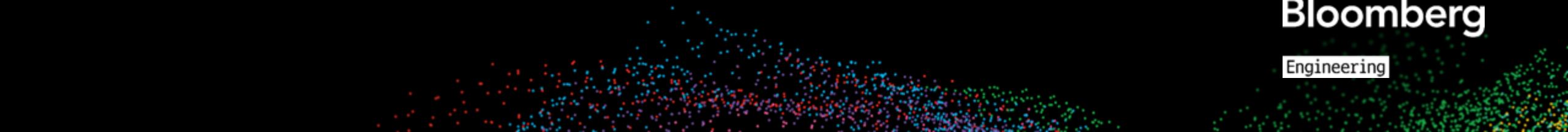
Address sanitizers in Rust/C++: example

Stack overflow is caught

```
==3490050==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7be09e9de120 …
READ of size 1 at 0x7be09e9de120 thread T0
    #0 0x7fe0a0ca115c  (/usr/local/lib/libfoo.so+0x515c)
    #1 0x556536122ccf  (rustapp/target/x86_64-unknown-linux-gnu/debug/rustapp+0x101ccf)
```

Nightly compiler

Unstable sanitizer feature



Bloomberg

Engineering

Address sanitizers in Rust/C++: Build 1/2

Build and integration are tricky

cxx build is done via custom build.rs script

Instrument bridge (flag)

Instrument app code (RUSTFLAGS)

```
# Cargo.toml
[package]
name = "rustapp"
edition = "2024"

[dependencies]
cxx = "1.0"

[build-dependencies]
cxx-build = "1.0"
```

Invocation: `// RUSTFLAGS="-Z sanitizer=address" cargo build --target x86_64-unknown-linux-gnu`

```
// build.rs
fn main() {
    cxx_build::bridge("src/main.rs").flag("-fsanitize=address").compile("cxxbridge");

    ...
}
```

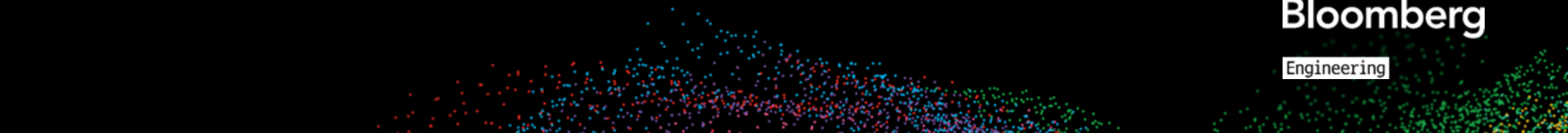
Address sanitizers in Rust/C++: Build 2/2

Only link existing ASan static runtime (same as C++ library)

Do not tell linker -fsanitize=address

```
// build.rs
fn main() {
    cxx_build::bridge("src/main.rs").flag("-fsanitize=address").compile("cxxbridge");

    println!("cargo:rustc-link-lib=foo");
    println!("cargo:rustc-link-lib=asan");
   // NOT: println!(cargo:rustc-link-arg=-fsanitize=address”);
}
```

Rust own ASan librustc-nightly_rt.asan.a runtime is not compatible with LLVM:
==82818==Your application is linkedagainst incompatible ASan runtimes



Bloomberg
Engineering

Address sanitizers in Rust/C++: summary

Rust is built upon LLVM

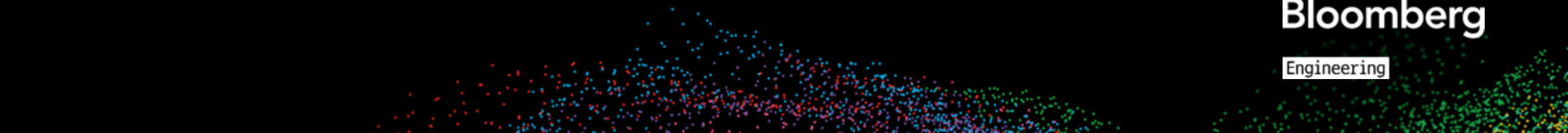
Reuse LLVM sanitizer runtime and LLVM compiler back-end

Different binary of runtime (GCC vs LLVM vs Rust)

Same sanitizers:

- Heap
- Stack
- Leaks

Cross-language memory tracking is real



Bloomberg

Engineering

Use

Ok, we are convinced to use sanitizers

But how?

And where?

Bloomberg

Engineering

Using sanitizers: Unit tests

Run unit tests

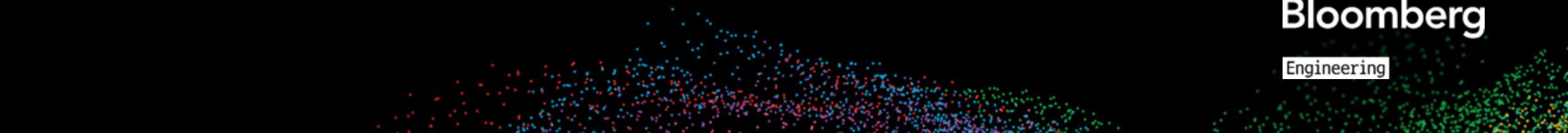
Mandatory

Good if code coverage is enough

- Code paths are covered

Good if data coverage is enough

- It is rare
- Bad for data-driven applications

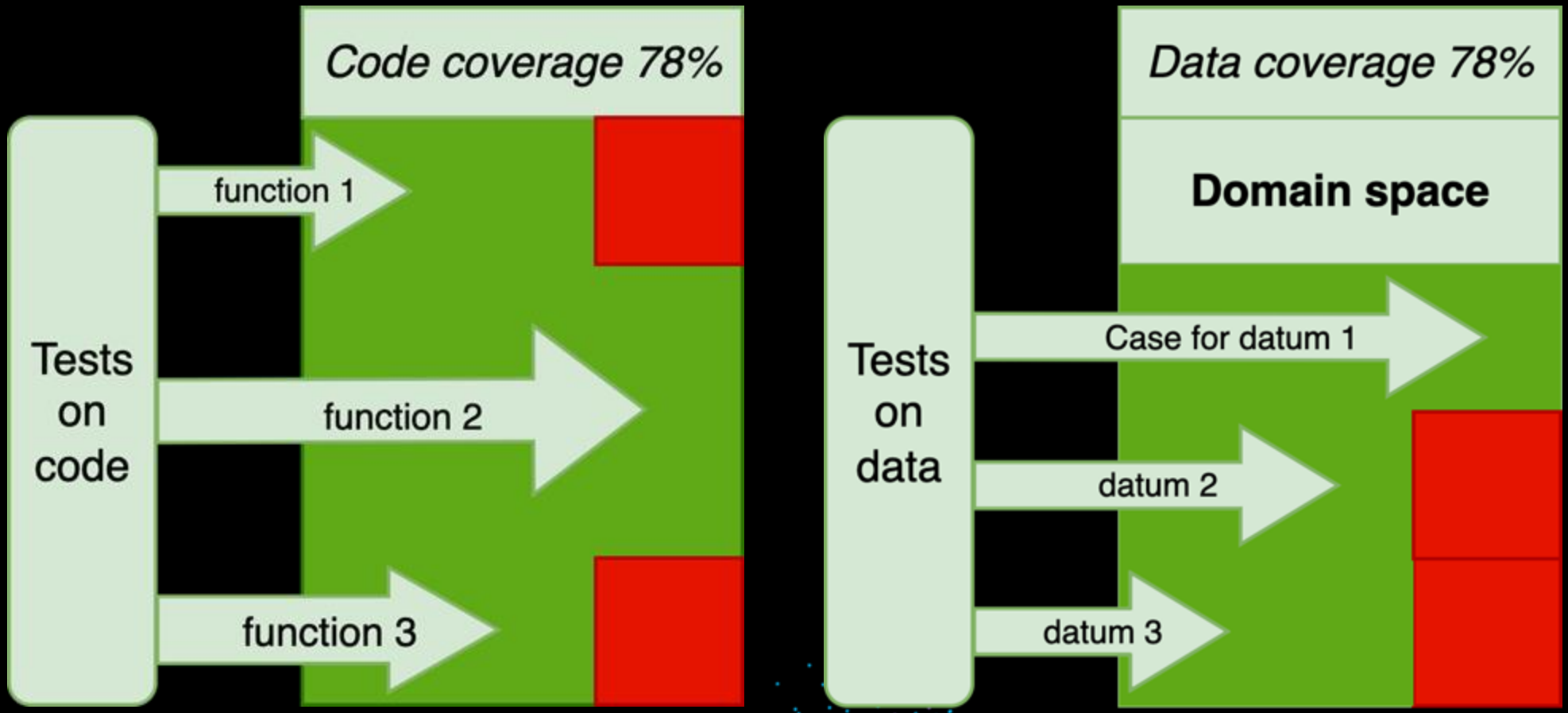


Bloomberg

Engineering

Using sanitizers: A problem with coverage

Coverage is not only for the code!



Bloomberg

Engineering

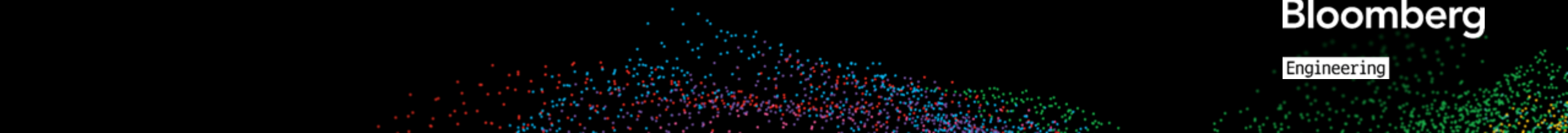
Using sanitizers: Integration tests

Run integration tests

More code paths exposed to sanitizers

Closer to real scenarios

Measure coverage for integration tests



Bloomberg

Engineering

Using sanitizers: Fuzzing

Coverage-guided runs of black-box program with mutating input data

Akin to property-based testing (white box)

✅ Good scenarios — structured input, stateless:

- Library
- Database
- Codec

⚠️ Bad scenarios — hard to isolate state:

- HTTP or RPC server
- Event-based systems
- Raw network monitoring

LLVM libfuzzer

Feeds mutating data

Provide an entrypoint

```
// fuzz_entry.cxx
extern "C" int LLVMFuzzerTestOneInput(
        const uint8_t *Data, size_t Size)
{
  RunSanitizedFunctonsOnInput(Data, Size);
  return 0;
}
```

Bloomberg

Engineering

Using sanitizers: Production

Is it safe? NO!

Most sanitizers are stable, some are beta

Not intended to be production ready

Reduce problem surface — minimal runtime

Safety hierarchy (higher to lower):

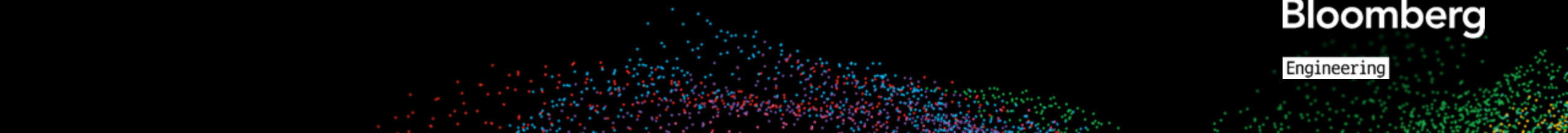
- Undefined
- Address
- Leak
- Thread
- Memory

Bloomberg

Engineering

Using sanitizers: Production

- Fail or continue? Recovery strategies
- Address sanitizer
  - Compile flag -fsanitize-recover=all
  - Runtime ASAN_OPTIONS=halt_on_error=0
- Undefined sanitizer
  - Just reports by default
  - Trap and continue -fsanitize-undefined-trap-on-error
- Thread sanitizer
  - By default, just reports
- Leak sanitizer
  - On exit
  - Can enable periodic reports (see above)



Bloomberg

Engineering

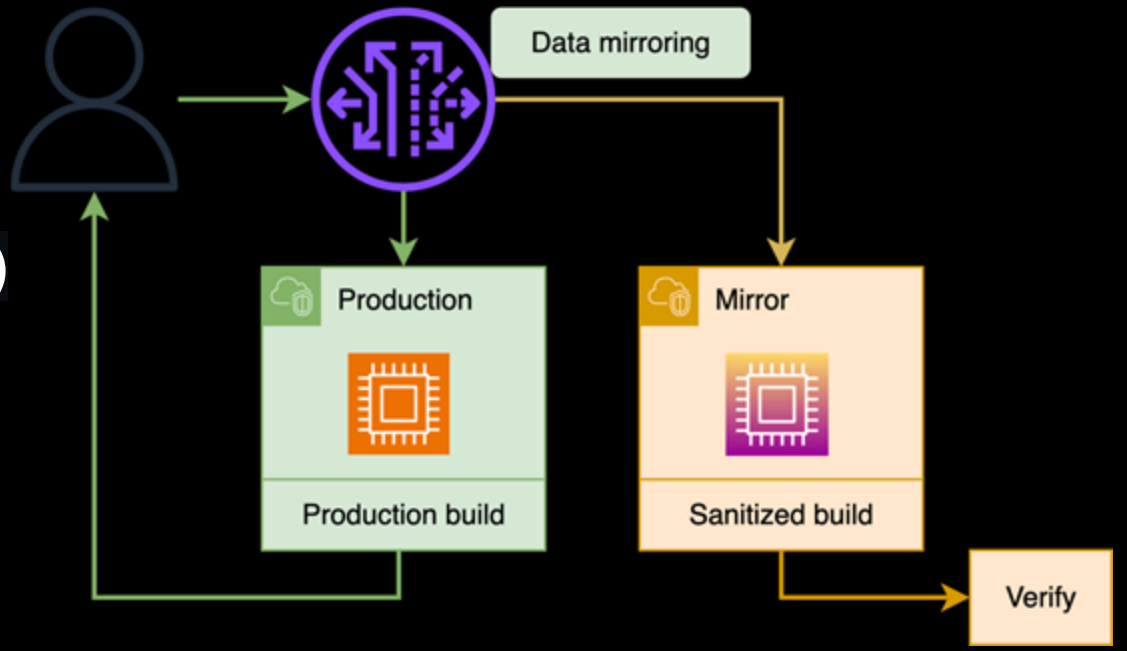
Using sanitizers: Production

Blue/green or canary deployment

- Evaluate the risks
- Real environment
- Part of user traffic goes to test instance

Mirroring data traffic

- Safe
- Must be fully isolated (mock or deployment)
- Output reconciliation



Bloomberg

Engineering

Wrap-up

Sanitizers are powerful tools

Available for major compilers and platforms

Open source implementations

Complex interaction at component boundaries

Tricky integration with build system

Better to run closer to the real system

Bloomberg

Engineering

Thank you!

Bloomberg

Engineering

TechAtBloomberg.com