
Core C++ 2026
20-21Oct. 2026 :: Tel-Aviv
Why Your Tests Still Miss Runtime Bugs
Alex Kushnir

CORE C++ 2026 // DYNAMIC ANALYSIS DEEP DIVE

Why Your Tests Still Miss Runtime Bugs

Valgrind, Compiler Sanitizers, and Runtime Debugging in Practice

SPEAKER

Alex Kushnir



SPEAKER PROFILE

About Me

BACKGROUND & EXPERIENCE

Software Engineer

Building complex systems since 2007

CURRENT ROLE

J&J MedTech | Electrophysiology

Developing software for cardiac mapping and ablation systems

KNOWLEDGE SHARING

Teaching Practical Workshops

Hands-on training focused on modern debugging techniques

engineer_profile.cpp

```cpp
// Core engineering focus
struct EngineerProfile {
    std::vector<std::string> focus = {
        "C++",
        "Build Systems",
        "Debugging Tools & Workflows",
        "Embedded Software"
    };

    // Problem-solving philosophy
    Workflow mission() {
        // hard bugs → repeatable workflows
        return Workflow::Automate(
            "complex_runtime_bugs");
    }
};
```

PRESENTATION STRUCTURE

Agenda

00 Opening

01 Why Tests Miss Runtime Bugs

02 Kinds of Runtime Bugs

03 How Dynamic Analysis Works

04 Valgrind

05 AddressSanitizer (ASan)

06 ThreadSanitizer (TSan)

07 UndefinedBehaviorSanitizer (UBSan)

08 Side-by-Side Comparison

09 A Practical Workflow

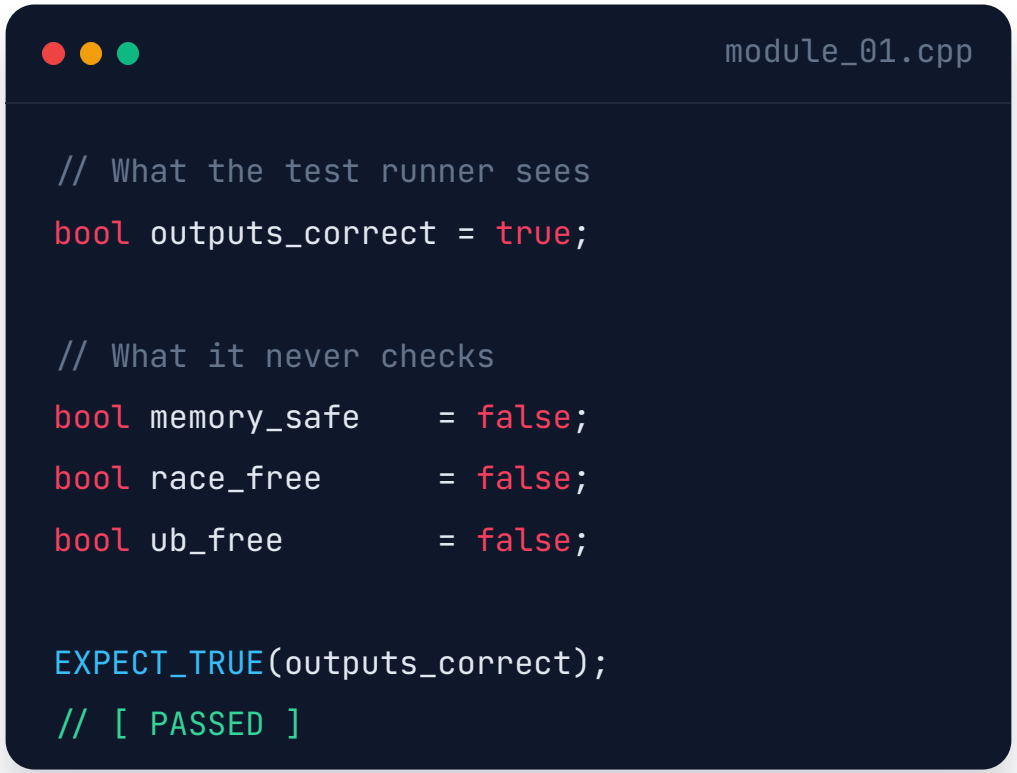
10 Takeaways

Q&A Questions

New to sanitizers? Which tool to run, and how. **Already using them?** Blind spots, custom allocators and the CI workflow.

Why Tests Miss Runtime Bugs

Green test suites, real failures, and the gap between them



THE TESTING PARADOX

Ready to Ship

THE TESTING PARADOX

Ready to Ship



Code Review

Approved by two reviewers

THE TESTING PARADOX

Ready to Ship

Code Review

Approved by two reviewers

Unit Tests

Thousands of tests passed, high coverage

THE TESTING PARADOX

Ready to Ship

Code Review

Approved by two reviewers

Unit Tests

Thousands of tests passed, high coverage

Integration

Nightly system tests green

THE TESTING PARADOX

Ready to Ship?

Code Review

Approved by two reviewers

Unit Tests

Thousands of tests passed, high coverage

Integration

Nightly system tests green

Production

Rare, silent data corruption under real load

THE TESTING PARADOX

Ready to Ship?

Code Review
Approved by two reviewers

Unit Tests
Thousands of tests passed, high coverage

Integration
Nightly system tests green

Production
Rare, silent data corruption under real load

> WAR STORY
>
> A data pipeline passed every test, then rejected a message every few days in production with a bad checksum. Finding it took days, and the logs never helped.

WAR STORY

The Checksum That Failed Every Few Days

WAR STORY

The Checksum That Failed Every Few Days

Symptom

A bad checksum once every few days under real load. No crash. Every test green, even long load runs.

WAR STORY

The Checksum That Failed Every Few Days

Symptom

A bad checksum once every few days under real load. No crash. Every test green, even long load runs.

False Leads

Hardware, then the transport, then the compiler (only at -O2). More logging made it rarer.

WAR STORY

The Checksum That Failed Every Few Days

Symptom

A bad checksum once every few days under real load. No crash. Every test green, even long load runs.

False Leads

Hardware, then the transport, then the compiler (only at -O2). More logging made it rarer.

ASan Saw Nothing

The pool reused its own blocks: no free to catch. With new/delete under ASan, the first run failed.

WAR STORY

The Checksum That Failed Every Few Days

Symptom

A bad checksum once every few days under real load. No crash. Every test green, even long load runs.

False Leads

Hardware, then the transport, then the compiler (only at -O2). More logging made it rarer.

ASan Saw Nothing

The pool reused its own blocks: no free to catch. With new/delete under ASan, the first run failed.

Root Cause

The producer queued a message, then returned it to the pool, which reused it while the worker read it.

WAR STORY

The Checksum That Failed Every Few Days

Symptom

A bad checksum once every few days under real load. No crash. Every test green, even long load runs.

False Leads

Hardware, then the transport, then the compiler (only at -O2). More logging made it rarer.

ASan Saw Nothing

The pool reused its own blocks: no free to catch. With new/delete under ASan, the first run failed.

Root Cause

The producer queued a message, then returned it to the pool, which reused it while the worker read it.

The fix: ownership moves with the message. Only the consumer releases it.

`Full example is available at the talk repo (00_war_story_intro)`

WAR STORY

The Bug: One Line Too Many

producer: RunPipeline()

```
157 m->m_id = static_cast<std::uint32_t>(i);
158 for (auto& b : m->m_payload)
159 {
160     b = static_cast<unsigned char>(i);
161 }
162
163 m->m_checksum = ChecksumOf(*m);
164 queue.Push(m);
165 pool.Release(m); // Bug: the worker still owns this message
```

worker thread

```
138 while (Message* m = queue.Pop())
139 {
140     std::this_thread::yield();          // simulate some work
141     if (ChecksumOf(*m) != m->m_checksum) // reads the message
142     {
143         ++stats.m_corrupted;
144     }
145     ++stats.m_processed;
146 }
```

The fix: delete line 165. The worker releases the message after it reads it.

Push() hands m to the worker. Release() hands the same buffer back to the producer, several messages later.

Full example is available at the talk repo (00_war_story_intro)

Compiler Explorer: no sanitizer · ASan · TSan

CORE PROBLEM

Why Test Suites Miss Runtime Bugs

CORE PROBLEM

Why Test Suites Miss Runtime Bugs

✓

Outputs, Not Invariants

Asserts compare return values. A program can corrupt its heap and still return 42.

CORE PROBLEM

Why Test Suites Miss Runtime Bugs

Outputs, Not Invariants

Asserts compare return values. A program can corrupt its heap and still return 42.

UB Is Silent

Undefined behavior often "works" at -O0, then breaks with -O2, a new compiler or a new target.

CORE PROBLEM

Why Test Suites Miss Runtime Bugs

Outputs, Not Invariants

Asserts compare return values. A program can corrupt its heap and still return 42.

UB Is Silent

Undefined behavior often "works" at -O0, then breaks with -O2, a new compiler or a new target.

Timing Is Hidden

Idle CI machines give friendly thread schedules. Races appear 1 run in 10,000, or only under load.

CORE PROBLEM

Why Test Suites Miss Runtime Bugs

Outputs, Not Invariants

Asserts compare return values. A program can corrupt its heap and still return 42.

UB Is Silent

Undefined behavior often "works" at -O0, then breaks with -O2, a new compiler or a new target.

Timing Is Hidden

Idle CI machines give friendly thread schedules. Races appear 1 run in 10,000, or only under load.

Cause Far From Symptom

Corruption happens in module A, but the crash surfaces minutes later in unrelated module B.

CORE PROBLEM

Why Test Suites Miss Runtime Bugs

Outputs, Not Invariants

Asserts compare return values. A program can corrupt its heap and still return 42.

UB Is Silent

Undefined behavior often "works" at -O0, then breaks with -O2, a new compiler or a new target.

Timing Is Hidden

Idle CI machines give friendly thread schedules. Races appear 1 run in 10,000, or only under load.

Cause Far From Symptom

Corruption happens in module A, but the crash surfaces minutes later in unrelated module B.

We need tools that watch how the program executes, not just what it returns.

EXAMPLE

Green Test, Off-by-One Overflow

heap_overflow.cpp

```cpp
#include <gtest/gtest.h>
#include <string>

std::string MakeLabel(const std::string& name)
{
    char* buf = new char[name.size()]; // no +1
    for (std::size_t i = 0; i ≤ name.size(); ++i)
        buf[i] = name[i]; // copies '\0' too

    std::string label(buf);
    delete[] buf;

    return label;
}

TEST(Label, CopiesName)
{
    EXPECT_EQ(MakeLabel("sensor"), "sensor");
}
```

same test, -fsanitize=address

```
==1617==ERROR: AddressSanitizer:
  heap-buffer-overflow on 0x502000000016
WRITE of size 1 at 0x502000000016 thread T0
  #0 MakeLabel(...) heap_overflow.cpp:8
  #1 TestBody() heap_overflow.cpp:18
0x502000000016 is located 0 bytes
  after 6-byte region
allocated by thread T0 here:
  #0 operator new[](unsigned long)
  #1 MakeLabel(...) heap_overflow.cpp:6
SUMMARY: AddressSanitizer:
  heap-buffer-overflow heap_overflow.cpp:8
```

Full example is available at the talk repo (01_heap_overflow)

Compiler Explorer: no sanitizer · ASan

WHAT CAN GO WRONG

Kinds of Runtime Bugs

WHAT CAN GO WRONG

Kinds of Runtime Bugs

MEMORY

- Heap, stack & global overflow
- Use-after-free
  / -return / -scope
- Double & invalid free
- Uninitialized reads
- Leaks

WHAT CAN GO WRONG

Kinds of Runtime Bugs

MEMORY

- Heap, stack & global overflow
- Use-after-free / -return / -scope
- Double & invalid free
- Uninitialized reads
- Leaks

CONCURRENCY

- Data races
- Lock-order inversion
- Deadlocks
- Unsafe publication of objects
- Destruction while in use by another thread

WHAT CAN GO WRONG

Kinds of Runtime Bugs

MEMORY

- Heap, stack & global overflow
- Use-after-free / -return / -scope
- Double & invalid free
- Uninitialized reads
- Leaks

CONCURRENCY

- Data races
- Lock-order inversion
- Deadlocks
- Unsafe publication of objects
- Destruction while in use by another thread

UNDEFINED BEHAVIOR

- Signed integer overflow
- Out-of-range shifts
- Misaligned & null access
- Invalid bool / enum values
- Wrong dynamic type (bad downcast)

WHAT CAN GO WRONG

Kinds of Runtime Bugs

MEMORY

- Heap, stack & global overflow
- Use-after-free / -return / -scope
- Double & invalid free
- Uninitialized reads
- Leaks

CONCURRENCY

- Data races
- Lock-order inversion
- Deadlocks
- Unsafe publication of objects
- Destruction while in use by another thread

UNDEFINED BEHAVIOR

- Signed integer overflow
- Out-of-range shifts
- Misaligned & null access
- Invalid bool / enum values
- Wrong dynamic type (bad downcast)

Each bug class needs its own detector, so no single tool catches them all.

TOOL COVERAGE

Which Tool Catches What

Bug class	Valgrind	ASan	TSan	UBSan
Heap overflow, use-after-free	Yes	Yes		
Stack & global overflow	No	Yes		
Uninitialized reads	Yes	No		
Leaks	Yes	Yes (LSan)		
Data races, lock order	Helgrind		Yes	
Overflow, shifts, alignment				Yes

No single column covers everything.

RUNTIME INSTRUMENTATION: TWO APPROACHES

How Dynamic Analysis Works

RUNTIME INSTRUMENTATION: TWO APPROACHES

How Dynamic Analysis Works

DYNAMIC TRANSLATION

Valgrind

No recompilation: runs the binary you ship.

Synthetic CPU: machine code → VEX IR → instrumented → JIT back to native.

Shadow state: validity bit per data bit, addressability bit per byte.

Cost: 10–50× slower, threads serialized.

RUNTIME INSTRUMENTATION: TWO APPROACHES

How Dynamic Analysis Works

DYNAMIC TRANSLATION

Valgrind

No recompilation: runs the binary you ship.

Synthetic CPU: machine code → VEX IR → instrumented → JIT back to native.

Shadow state: validity bit per data bit, addressability bit per byte.

Cost: 10–50× slower, threads serialized.

INLINE INSTRUMENTATION

GCC / Clang / MSVC Sanitizers

Compiler-injected: checks around memory accesses and risky operations.

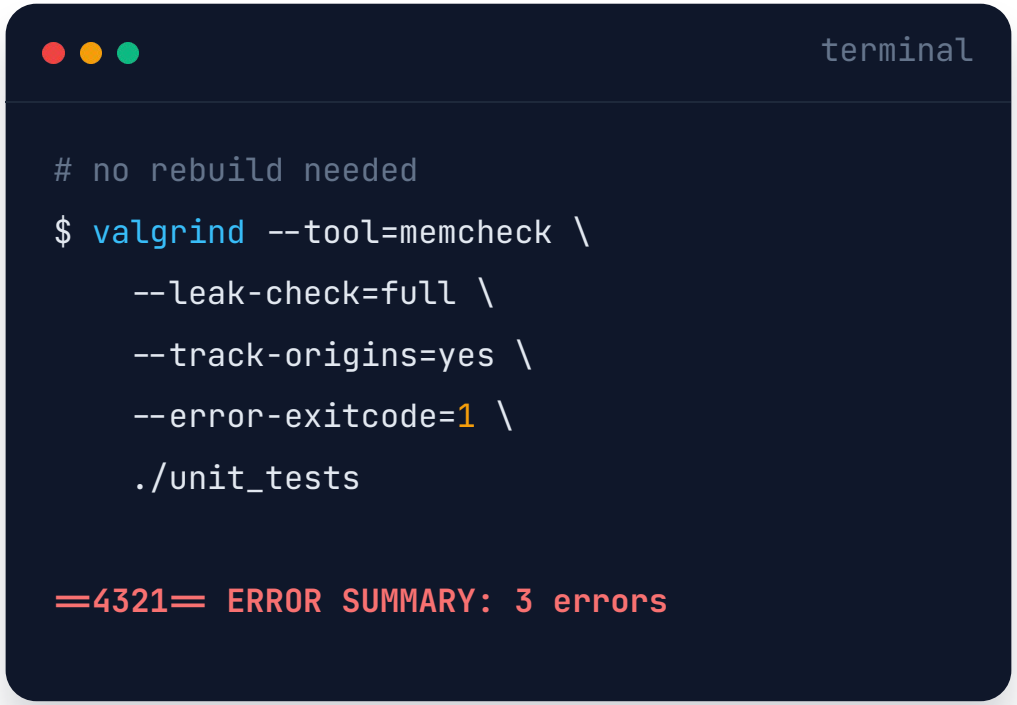
Runtime library: intercepts malloc/free, pthreads, libc.

Shadow memory: e.g. ASan maps 8 bytes → 1 shadow byte.

Cost: ~2× (ASan) to 5–15× (TSan); needs a rebuild.

Valgrind

Memcheck and its reports, the tool family, strengths and blind spots



TOOL DEEP DIVE: MEMCHECK

Memcheck: Tracking Every Bit

What it reports

- Invalid reads and writes on the heap
- Uninitialized values that affect a branch or an address
- Invalid and mismatched frees (new → free(), new[] → delete)
- Overlapping memcpy source and destination
- Leaks: memory never released (new without delete)

parse.cpp — valgrind

```
 8 int Parse(const char* buf)
 9 {
10     Header h; // not zeroed
11     std::memcpy(&h.m_len, buf, 4);
12     if (h.m_flags & 0x1) // uninit!
13         return -1;
14     return h.m_len;
15 }

==1586437== Conditional jump or move depends
==1586437==   on uninitialised value(s)
==1586437==    at 0x109179: Parse(char const*) (parse.cpp:12)
==1586437==    by 0x1091C8: main (parse.cpp:20)
```

THE CODE

The Code Behind the Report

Server.cpp

```
3  void CloseIdle(SessionPool& pool,
                  Request& req)
4  {
5      pool.Release(req.m_session); // dangling
6  }
8  int Handle(Request& req)
9  {
10     return req.m_session->Id(); // ①
11 }
```

SessionPool.cpp

```
3  Session* SessionPool::Acquire(int id)
4  {
5      return new Session(id); // ③ alloc'd here
6  }
8  void SessionPool::Release(Session* s)
9  {
10     delete s; // ② free'd here
11 }
```

Session.cpp

```
5  int Session::Id() const
6  {
7      return m_id; // ① read of size 4
8  }
```

Full example is available at the talk repo (03_parsing_valgrind_output)

READING THE OUTPUT

Anatomy of a Memcheck Report

valgrind ./parsing_tests

```
==1279973== Invalid read of size 4                                    ①
==1279973==    at 0x113502: Session::Id() const (Session.cpp:7)
==1279973==    by 0x1135EC: Handle(Request&) (Server.cpp:10)
==1279973==  Address 0x4e33bf0 is 16 bytes inside a block             ②
==1279973==    of size 64 free'd
==1279973==    at 0x484A61D: operator delete(void*, unsigned long)
==1279973==    by 0x11359D: SessionPool::Release(Session*) (SessionPool.cpp:10)
==1279973==    by 0x1135CA: CloseIdle(SessionPool&, Request&) (Server.cpp:5)
==1279973==  Block was alloc'd at                                      ③
==1279973==    at 0x4846FA3: operator new(unsigned long)
==1279973==    by 0x113529: SessionPool::Acquire(int) (SessionPool.cpp:5)
```

Full example is available at the talk repo (03_parsing_valgrind_output)

① What & where

Access kind, size, and the stack of the bad access.

② Which memory, who freed it

“Inside a block … free'd” means use-after-free, the stack shows who freed it.

③ Who allocated it

Which object it was, and who owns its lifetime.

THE CODE

Four Ways to Leak Memory

THE CODE

Four Ways to Leak Memory

① DEFINITELY LOST

```
19 void DefinitelyLost()
20 {
21     int* samples = new int[16];
22     samples[0] = 1;
23 } // 'samples' goes out of scope
```

THE CODE

Four Ways to Leak Memory

① DEFINITELY LOST

```
19 void DefinitelyLost()
20 {
21     int* samples = new int[16];
22     samples[0] = 1;
23 } // 'samples' goes out of scope
```

② INDIRECTLY LOST

```
27 void IndirectlyLost()
28 {
29     Node* root = new Node;
30     root->m_child = new Node;
31 } // root: definitely lost, child: indirectly lost
```

THE CODE

Four Ways to Leak Memory

① DEFINITELY LOST

```
19 void DefinitelyLost()
20 {
21     int* samples = new int[16];
22     samples[0] = 1;
23 } // 'samples' goes out of scope
```

② INDIRECTLY LOST

```
27 void IndirectlyLost()
28 {
29     Node* root = new Node;
30     root→m_child = new Node;
31 } // root: definitely lost, child: indirectly lost
```

③ POSSIBLY LOST

```
16 char* gCursor = nullptr; // points into the MIDDLE of a block
34 void PossiblyLost()
35 {
36     char* buffer = static_cast<char*>(std::malloc(64));
37     std::memset(buffer, 0, 64);
38     gCursor = buffer + 16; // no pointer to the start is kept
39 }
```

THE CODE

Four Ways to Leak Memory

① DEFINITELY LOST

```
19 void DefinitelyLost()
20 {
21     int* samples = new int[16];
22     samples[0] = 1;
23 } // 'samples' goes out of scope
```

② INDIRECTLY LOST

```
27 void IndirectlyLost()
28 {
29     Node* root = new Node;
30     root→m_child = new Node;
31 } // root: definitely lost, child: indirectly lost
```

③ POSSIBLY LOST

```
16 char* gCursor = nullptr; // points into the MIDDLE of a block
34 void PossiblyLost()
35 {
36     char* buffer = static_cast<char*>(std::malloc(64));
37     std::memset(buffer, 0, 64);
38     gCursor = buffer + 16; // no pointer to the start is kept
39 }
```

④ STILL REACHABLE

```
15 char* gConfig = nullptr; // points to the START of a block
42 void StillReachable()
43 {
44     gConfig = static_cast<char*>(std::malloc(128));
45     std::strcpy(gConfig, "log_level=debug");
46 }
```

Full example is available at the talk repo (04_leak_kinds)

READING THE OUTPUT

Anatomy of a Leak Report

```
                                        --show-leak-kinds=all ./leak_kinds
==1589033== 32 bytes in 1 blocks are indirectly lost
==1589033==    by 0x1091FF: IndirectlyLost() (leak_kinds.cpp:30)
==1589033== 64 bytes in 1 blocks are possibly lost
==1589033==    by 0x10922E: PossiblyLost() (leak_kinds.cpp:36)
==1589033== 64 bytes in 1 blocks are definitely lost
==1589033==    by 0x1091BE: DefinitelyLost() (leak_kinds.cpp:21)
==1589033== 64 (32 direct, 32 indirect) bytes in 1 blocks
==1589033==  are definitely lost
==1589033==    by 0x1091E6: IndirectlyLost() (leak_kinds.cpp:29)
==1589033== 128 bytes in 1 blocks are still reachable
==1589033==    by 0x10926C: StillReachable() (leak_kinds.cpp:44)
==1589033== LEAK SUMMARY:
==1589033==    definitely lost: 96 bytes in 2 blocks
==1589033==    indirectly lost: 32 bytes in 1 blocks
==1589033==      possibly lost: 64 bytes in 1 blocks
==1589033==    still reachable: 128 bytes in 1 blocks
```

① Definitely lost
No pointer remains. A real leak: fix now.

② Indirectly lost
Reachable only via a lost block. Fix the owner and it goes too.

③ Possibly lost
Only a pointer into the middle survives. Investigate: pools, tagged pointers.

④ Still reachable
Still referenced at exit, e.g. a global. Agree on a team policy.

Full example is available at the talk repo (04_leak_kinds)

BEYOND MEMCHECK

One Framework, Many Tools

BEYOND MEMCHECK

One Framework, Many Tools

`--tool=helgrind`

Helgrind

Data races, lock-order problems and pthread API misuse.

BEYOND MEMCHECK

One Framework, Many Tools

`--tool=helgrind`

Helgrind

Data races, lock-order problems and pthread API misuse.

`--tool=drd`

DRD

Races and lock contention, with lower memory use than Helgrind.

BEYOND MEMCHECK

One Framework, Many Tools

`--tool=helgrind`

Helgrind

Data races, lock-order problems and pthread API misuse.

`--tool=drd`

DRD

Races and lock contention, with lower memory use than Helgrind.

`--tool=massif`

Massif

Heap profiler: who holds how much memory, over time.

BEYOND MEMCHECK

One Framework, Many Tools

`--tool=helgrind`

Helgrind

Data races, lock-order problems and pthread API misuse.

`--tool=drd`

DRD

Races and lock contention, with lower memory use than Helgrind.

`--tool=massif`

Massif

Heap profiler: who holds how much memory, over time.

`--tool=callgrind`

Callgrind

Call graphs and instruction counts; Cachegrind adds cache simulation.

BEYOND MEMCHECK

One Framework, Many Tools

`--tool=helgrind`

Helgrind

Data races, lock-order problems and pthread API misuse.

`--tool=drd`

DRD

Races and lock contention, with lower memory use than Helgrind.

`--tool=massif`

Massif

Heap profiler: who holds how much memory, over time.

`--tool=callgrind`

Callgrind

Call graphs and instruction counts; Cachegrind adds cache simulation.

Same binary, same command line, different argument: `--tool=<name>`.

VALGRIND TRADE-OFFS

Strengths and Blind Spots

STRENGTHS

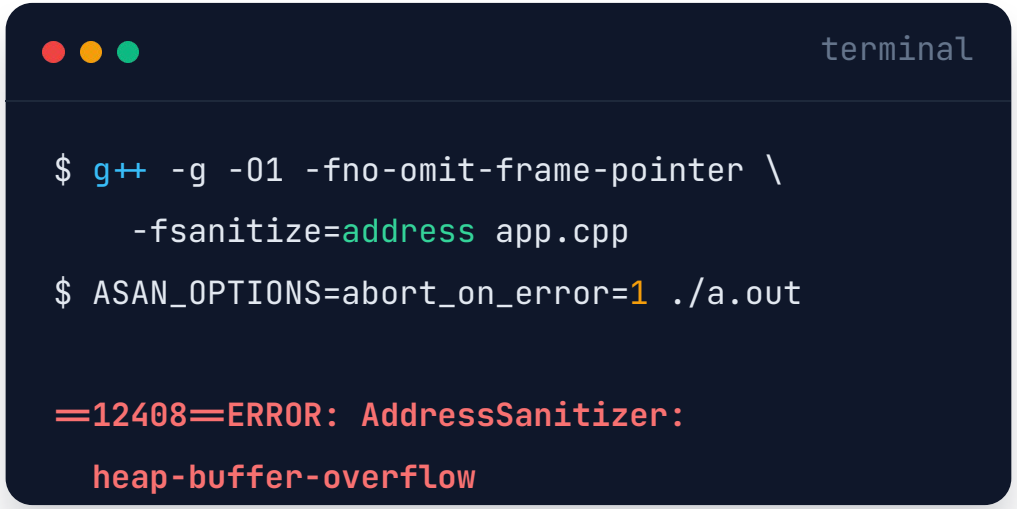
- No rebuild: test release and third-party binaries
- Best-in-class uninitialized-value tracking
- Mature suppressions and client requests
- vgdb: stop in gdb at the first error

BLIND SPOTS & COSTS

- 10-50× slower, threads run one at a time
- No bounds checks on stack and global arrays
- Custom allocators invisible without VALGRIND_MALLOCLIKE_BLOCK / MAKE_MEM_* annotations
- Mostly Linux (macOS way behind, no Windows)

AddressSanitizer

Heap, stack and global overflows, use-after-free and leaks, at about 2× cost



TOOL DEEP DIVE: ASAN

AddressSanitizer: What It Catches

Heap overflow

Reads and writes past a malloc/new block.

Stack/global overflow

Redzones around locals and globals too.

Use-after-free

Freed memory is quarantined, not reused right away.

Use-after-return

And use-after-scope: references to dead locals.

Double & invalid free

Plus new/delete[] mismatches.

LeakSanitizer (LSan)

At exit, reports heap blocks nothing points to. On by default on Linux.

About 2× CPU and 2-3× memory, GCC ≥ 4.8, Clang ≥ 3.1, MSVC ≥ 16.9, almost no false positives.

UNDER THE HOOD

Shadow Memory, Redzones, Quarantine

MEMORY: new char[16] with its redzones, and the same bytes after delete[]

redzone	new char[16]: 16 usable bytes	redzone	after delete[]
fa fa	00 00	fa fa	fd fd

↑ SHADOW: 1 byte describes each 8 bytes of memory above

Reading the shadow: 00 = all 8 bytes usable; 01–07 = only the first N, when the requested size ends mid-group; fa = redzone; fd = freed.

Redzones surround every allocation, **quarantine** delays reuse of freed blocks, so stale pointers still land on fd.

```
// check inserted before a 4-byte (int) load
int8_t k = *(int8_t*)((addr >> 3) + kOffset);
if (k ≠ 0 && (addr & 7) + 3 ≥ k)
    __asan_report_load4(addr);
```

THE CODE

Off-by-One in a Ring Buffer

RingBuffer.cpp

```cpp
RingBuffer::RingBuffer(std::size_t capacity)
    : m_data(new int[capacity]), m_capacity(capacity) {}

…

void RingBuffer::Push(int value)
{
    m_data[m_head] = value; // writes m_data[m_capacity] once per lap
    ++m_pushed;
    if (++m_head > m_capacity) // BUG: should be ⩾
    {
        m_head = 0;
    }
}
```

Producer.cpp

```cpp
void Producer::Run(int count)
{
    for (int i = 0; i < count; ++i)
    {
        m_ringBuffer.Push(i);
    }
}
```

RingBufferTest.cpp

```cpp
TEST(RingBuffer, ProducerFillsBuffer)
{
    RingBuffer ring(25); // 25 ints = 100 bytes
    Producer producer(ring);
    producer.Run(30); // wraps around once
    EXPECT_EQ(ring.Pushed(), 30u);
}
```

Full example is available at the talk repo (05_asan_ring_buffer_overflow)

Compiler Explorer: no sanitizer · ASan

READING THE OUTPUT

Anatomy of an ASan Report

./ring_buffer_tests

```
==1590832==ERROR: AddressSanitizer: heap-buffer-overflow
  on address 0x50b000000154 at pc 0x556a300ef94b
WRITE of size 4 at 0x50b000000154 thread T0                    ①
    #0 RingBuffer::Push(int) RingBuffer.cpp:12
    #1 Producer::Run(int) Producer.cpp:7
0x50b000000154 is located 0 bytes after                        ②
  100-byte region [0x50b0000000f0,0x50b000000154)
allocated by thread T0 here:                                   ③
    #0 operator new[](unsigned long)
    #1 RingBuffer::RingBuffer(unsigned long) RingBuffer.cpp:3
Shadow bytes around the buggy address:                         ④
=>0x50b000000100: 00 00 00 00 00 00 00 00 00 00[04]fa fa
  04 = 4 addressable bytes · fa = heap redzone
```

① Bug type & access
WRITE of 4 bytes, plus the stack that did it.

② Position
"0 bytes after a 100-byte region" = off-by-one.

③ Origin
Who sized the buffer: often where the fix goes.

④ Shadow bytes
[04]: only 4 of 8 bytes valid, you touched byte 100.

Where ASan Needs Your Help

Pool.cpp

```
 3 #include <sanitizer/asan_interface.h>
 …
22 void* Pool::Alloc()
23 {
24     void* p = m_free.back();
25     m_free.pop_back();
26     // handed out: make it accessible
27     ASAN_UNPOISON_MEMORY_REGION(p, sBlockSize);
28     return p;
29 }
30
31 void Pool::Release(void* p)
32 {
33     // returned: any access is now a bug
34     ASAN_POISON_MEMORY_REGION(p, sBlockSize);
35     m_free.push_back(p);
36 }
```

std::vector spare capacity

Reading past size() looks valid to ASan. libc++ catches it by default, libstdc++ needs -D_GLIBCXX_SANITIZE_VECTOR in every TU.

ASAN_OPTIONS worth setting

abort_on_error=1 core dump for gdb

detect_stack_use_after_return=1 dangling locals

check_initialization_order=1 global init order

Still not caught

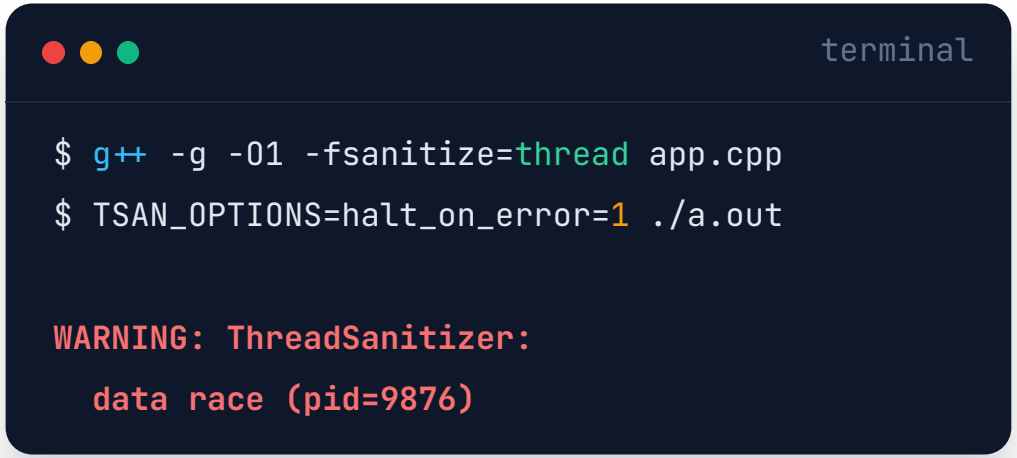
Uninitialized reads, overflow between fields of one object, code built without ASan.

Full example is available at the talk repo (06_asan_pool_poisoning)

Compiler Explorer: no sanitizer · ASan

ThreadSanitizer

Data races and lock-order inversions, found even when the run got lucky



TOOL DEEP DIVE: TSAN

ThreadSanitizer: Latent Races

Happens-before analysis: flags two accesses to the same memory that no lock, atomic or join orders, even if this run got lucky.

What it needs: both accesses must execute in the test. The bad interleaving does not have to happen.

Also finds: lock-order inversions (potential deadlocks), use of destroyed mutexes.

Cost: 5-15× CPU, 5-10× memory.

Supported since: Clang >= 3.2, GCC >= 4.8. 64-bit targets only.

Race.cpp — -fsanitize=thread

```
 1 bool stop = false; // plain bool
…
16 void Worker()
17 {
18     while (!stop)
19     {
20         Poll();
21     } // read
22 }
23
24 int main()
25 {
26     std::thread t{Worker};
27     std::this_thread::sleep_for(1s);
28     stop = true; // write: race
29     t.join();
30 }
31 // fix: std::atomic<bool> stop{false};
```

Full example is available at the talk repo (07_tsan_basic_example)

Compiler Explorer: no sanitizer · TSan

THE CODE

One Missing Lock

Stats.cpp

```
 3 void Stats::Record(int latencyMs)
 4 {
 5     std::lock_guard<std::mutex> lock(m_mutex);
 6     ++m_count;
 7     if (latencyMs > m_max)
 8     {
 9         m_max = latencyMs;
10     }
11 }
…
19 int Stats::Max() const
20 {
21     return m_max;    ◀ no lock
22 }
```

Worker.cpp

```
15 void Worker::Run(int requests)
16 {
17     for (int i = 0; i < requests; ++i)
18     {
19         m_stats.Record(i % 50); // 0..49 ms
20     }
21 }
```

StatsTest.cpp

```
 8 TEST(Stats, MaxWhileWorkerRuns)
 9 {
10     Stats stats;
11     Worker worker(stats);
12     worker.Start(1000);
13     int seen = stats.Max(); // races with Record
14     worker.Join();
15     EXPECT_LE(seen, 49);
16     EXPECT_EQ(stats.Max(), 49);
17 }
```

Full example is available at the talk repo (08_tsan_report)

Compiler Explorer: no sanitizer · TSan

READING THE OUTPUT

Anatomy of a TSan Report

./stats_tests

```
WARNING: ThreadSanitizer: data race (pid=1592750)
  Write of size 4 at 0x7ffd378bd470 by thread T1                    ①
    (mutexes: write M0):
    #0 Stats::Record(int) Stats.cpp:9
    #1 Worker::Run(int) Worker.cpp:19
  Previous read of size 4 at 0x7ffd378bd470 by main thread:
    #0 Stats::Max() const Stats.cpp:21
    #1 Stats_MaxWhileWorkerRuns_Test::TestBody() StatsTest.cpp:13
  Location is stack of main thread.                                 ②
  Mutex M0 (0x7ffd378bd440) created at:
    #4 Stats::Record(int) Stats.cpp:5
  Thread T1 (tid=1592752, running) created by main thread at:       ③
    #0 pthread_create
    #2 Stats_MaxWhileWorkerRuns_Test::TestBody() StatsTest.cpp:12
SUMMARY: ThreadSanitizer: data race Stats.cpp:9                     ④
  in Stats::Record(int)
```

① Two stacks
The write in T1 holds M0, read in main holds nothing.

② Location & mutex
Where the shared data lives and which mutex protects it.

③ Thread origins
Where T1 was started: StatsTest.cpp:12.

④ SUMMARY line
Stable and greppable: use it to dedupe in CI.

Full example is available at the talk repo (08_tsan_report)

TSAN TRADE-OFFS

Blind Spots & Annotations

LIMITS & FALSE REPORTS

- Races on paths your tests never run from two threads at once
- Custom sync it can't see (inline asm, raw futex): false reports until you annotate
- Uninstrumented libraries: false reports
- Standalone atomic_thread_fence: false reports (GCC warns with -Wtsan)
- Can't be combined with ASan or MSan

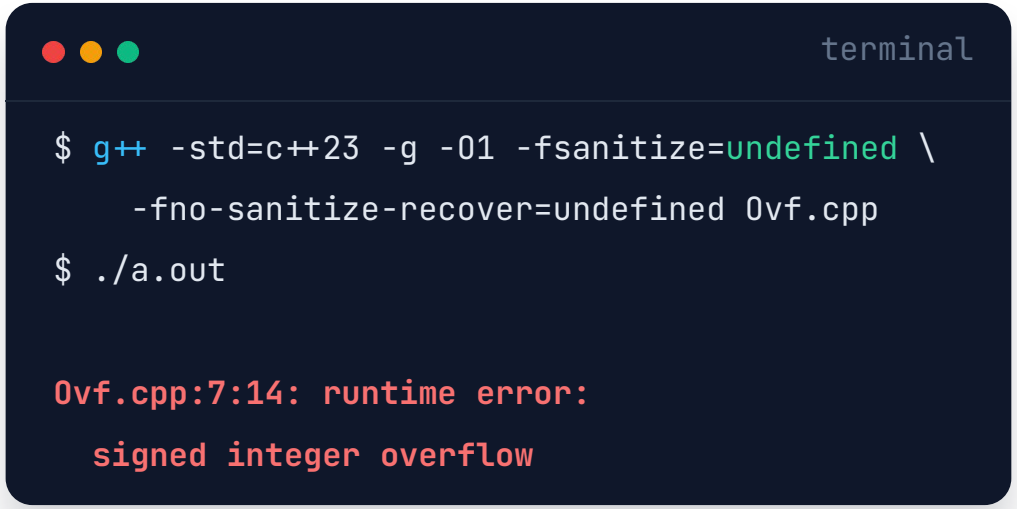
lockfree_stack.cpp

```
#include <sanitizer/tsan_interface.h>
// publish via handwritten asm
void publish(Node* n) {
    __tsan_release(&head_);
    asm_store_release(&head_, n);
}
Node* consume() {
    Node* n = asm_load_acquire(&head_);
    __tsan_acquire(&head_);
    return n;
}

# tsan.supp: last resort, with an owner
race:third_party/legacy_log/*
```

Undefined Behavior Sanitizer

Overflow, bad shifts, misalignment and more, for a low runtime cost



TOOL DEEP DIVE: UBSAN

UBSan: A Check per Operation

Inline checks: the compiler adds a check before each operation that can be UB: overflow, shifts, array index, null, alignment.

What it needs: the bad operation must run with the bad values. No shadow memory, no history.

Also finds: overflows inside one object, which ASan misses: an index past a member array.

Cost: usually under 2× CPU, almost no memory. Combines with ASan or TSan.

Supported since: Clang >= 3.3, GCC >= 4.9. Any target, including 32-bit and embedded.

Header.cpp — -fsanitize=undefined

```
4  struct Header
5  {
6      char m_tag[4];
7      std::uint32_t m_length;
8  };
…
10 void SetTag(Header& h, const char* tag)
11 {
12     for (int i = 0; i ⩽ 4; ++i)
13     { // copies the '\0' too
14         h.m_tag[i] = tag[i];
15     }
16 }
…
20     Header h{{}, 0x12345678};
21     SetTag(h, "DATA");
22     std::println("{:x}", h.m_length); // prints 12345600
24 // fix: i < 4 (tag is not a C string)
```

Full example is available at the talk repo (09_ubsan_basic_demo)

Compiler Explorer: no sanitizer · UBSan

TOOL DEEP DIVE: UBSAN

The Optimizer Deleted My Check

Ovf.cpp

```cpp
#include <limits>
#include <print>

[[nodiscard]]
bool WillOverflow(int x)
{
    return x + 1 < x;
}

int main()
{
    auto result = WillOverflow(std::numeric_limits<int>::max());
    std::println("Result is {}", result);
    return 0;
}
```

terminal

```
$ g++ -std=c++23 -O2 Ovf.cpp && ./a.out
Result is false

$ g++ -std=c++23 -O2 -fsanitize=undefined \
    Ovf.cpp && ./a.out
Ovf.cpp:7:14: runtime error: signed integer
  overflow: 2147483647 + 1 cannot be
  represented in type 'int'
Result is true
```

```
; g++ -O2                    ; g++ -O2 -fwrapv
xor  eax, eax                cmp  edi, 2147483647
ret                          sete al
                             ret
```

The fix

Check the input, not the result: return x == INT_MAX or let the compiler check the add: __builtin_add_overflow(x, 1, &sum).

Full example is available at the talk repo (10_ubsan_optimizer)

Compiler Explorer: no sanitizer · UBSan

UBSAN IN PRACTICE

Checks Worth Turning On

Check	Catches	Notes
signed-integer-overflow	INT_MAX + 1, negation of INT_MIN	The classic
shift	Shift ≥ bit width, negative shift	Common in protocol code
alignment, null	Misaligned loads, null member access	Packed structs, embedded
bool, enum	Loading an invalid bool or enum value	Often uninitialized memory
vptr	Call or cast on the wrong dynamic type	Needs RTTI
float-cast-overflow	1e20 converted to int	Opt-in on GCC
implicit-conversion	Lossy integer conversions (not UB)	Clang only, opt-in

-fsanitize=undefined turns on every row above except float-cast-overflow (on GCC) and implicit-conversion.

UBSAN IN PRACTICE

Making UBSan Fatal (or Trap)

```
-fsanitize=undefined -fno-sanitize-recover=undefined          # fatal: fail the test in CI
-fsanitize=undefined -fsanitize-trap=undefined                # trap mode: embedded, production
-fsanitize=undefined,float-cast-overflow -fno-sanitize=vptr   # add / remove checks
```

Trap mode: a failed check executes an illegal instruction (ud2 / brk) and crashes on the spot.
No report and no runtime library, so it fits firmware and release builds. Debug from the crash address.

THE REST OF THE FAMILY

Worth Knowing About

THE REST OF THE FAMILY

Worth Knowing About

`-fsanitize=memory`

MSan

Uninitialized reads at ~3× cost. Clang on Linux, every library, libc++ included, must be instrumented.

THE REST OF THE FAMILY

Worth Knowing About

`-fsanitize=memory`

MSan

Uninitialized reads at ~3× cost. Clang on Linux, every library, libc++ included, must be instrumented.

`-fsanitize=hwaddress`

HWASan

ASan-like detection using AArch64 pointer tags, with far less memory overhead.

THE REST OF THE FAMILY

Worth Knowing About

`-fsanitize=memory`

MSan

Uninitialized reads at ~3× cost. Clang on Linux, every library, libc++ included, must be instrumented.

`-fsanitize=hwaddress`

HWASan

ASan-like detection using AArch64 pointer tags, with far less memory overhead.

`sampling allocator`

GWP-ASan

Guards a tiny random sample of allocations in production, at negligible cost.

THE REST OF THE FAMILY

Worth Knowing About

`-fsanitize=memory`

MSan

Uninitialized reads at ~3× cost. Clang on Linux, every library, libc++ included, must be instrumented.

`-fsanitize=hwaddress`

HWASan

ASan-like detection using AArch64 pointer tags, with far less memory overhead.

`sampling allocator`

GWP-ASan

Guards a tiny random sample of allocations in production, at negligible cost.

`kernel config`

KASAN, KCSAN

Memory-error and race detection for Linux kernel and driver code.

THE REST OF THE FAMILY

Worth Knowing About

`-fsanitize=memory`

MSan

Uninitialized reads at ~3× cost. Clang on Linux, every library, libc++ included, must be instrumented.

`-fsanitize=hwaddress`

HWASan

ASan-like detection using AArch64 pointer tags, with far less memory overhead.

`sampling allocator`

GWP-ASan

Guards a tiny random sample of allocations in production, at negligible cost.

`kernel config`

KASAN, KCSAN

Memory-error and race detection for Linux kernel and driver code.

Same idea, different targets - pick the one that fits where your code runs.

SIDE BY SIDE

Strengths, Costs, Constraints

Tool	Detects	CPU	Memory	Rebuild	Toolchains
Valgrind	Heap errors, uninit, leaks	10-50×	High	No	Linux, any compiler
ASan	Heap/stack/global OOB, UAF, leaks	~2×	~2-3×	Yes	GCC, Clang, MSVC
TSan	Data races, lock order	5-15×	5-10×	Yes	GCC, Clang (64-bit)
UBSan	Overflow, shifts, alignment, types	Low	Low	Yes	GCC, Clang
MSan	Uninitialized reads	~3×	~2×	Yes, all code	Clang, Linux

Typical figures from the tool documentation, actual benchmarks may vary.

SIDE BY SIDE

Strengths, Costs, Constraints

Tool	Detects	CPU	Memory	Rebuild	Toolchains
Valgrind	Heap errors, uninit, leaks	10-50×	High	No	Linux, any compiler
ASan	Heap/stack/global OOB, UAF, leaks	~2×	~2-3×	Yes	GCC, Clang, MSVC
TSan	Data races, lock order	5-15×	5-10×	Yes	GCC, Clang (64-bit)
UBSan	Overflow, shifts, alignment, types	Low	Low	Yes	GCC, Clang
MSan	Uninitialized reads	~3×	~2×	Yes, all code	Clang, Linux

Typical figures from the tool documentation, actual benchmarks may vary.

No single tool wins - combine them across dev, CI and production.

COMBINING TOOLS

What Can Run Together

BUILD A

ASan + UBSan

Default debug and PR build. Covers most memory and UB bugs.

BUILD B

TSan + UBSan

Multithreaded tests and stress loops.

BUILD C (OPTIONAL)

MSan

Clang only, with every dependency rebuilt.

NO REBUILD

Valgrind

On the unmodified release binary, nightly.

> ASan, TSan and MSan **cannot share a binary** - **UBSan fits with all**.

A Practical Workflow

Putting the tools to work in development, CI and regression testing

ci.sh

```
# every pull request
cmake --preset asan-ubsan && ctest
cmake --preset tsan       && ctest -L mt

# nightly
valgrind --error-exitcode=1 \
    ./release/unit_tests
```

DEVELOPMENT

Make the Sanitized Build the Default

dev setup

```
# everyday debug build
CXXFLAGS="-O1 -g -fno-omit-frame-pointer \
  -fsanitize=address,undefined \
  -fno-sanitize-recover=undefined"
export ASAN_OPTIONS=abort_on_error=1
export UBSAN_OPTIONS=print_stacktrace=1
# stop in gdb at the first report
(gdb) break __asan::ReportGenericError
(gdb) rbreak ^__ubsan_handle_
# Valgrind + gdb
$ valgrind --vgdb=yes --vgdb-error=0 ./app
(gdb) target remote | vgdb
# rewind to the corruption
$ rr record ./app && rr replay
```

Fast enough for daily work

ASan + UBSan at ~2× is cheap enough for local builds and CI. Not for production builds.

Symbolized stacks

Keep -g. GCC symbolizes reports by itself; with Clang, put llvm-symbolizer on PATH.

Debugger at the report

Break on the report, or record with rr and step backwards.

CI: EVERY PR AND NIGHTLY

Every Tool Gets a Slot

EVERY PR

ASan + UBSan

Full unit and integration suite. Fails on the first report.

EVERY PR

TSan

Threaded tests, repeated (--gtest_repeat) to widen coverage.

NIGHTLY

Valgrind + long stress

Release binaries under Memcheck; longer TSan soak runs.

CI: EVERY PR AND NIGHTLY

Every Tool Gets a Slot

EVERY PR

ASan + UBSan

Full unit and integration suite. Fails on the first report.

EVERY PR

TSan

Threaded tests, repeated (--gtest_repeat) to widen coverage.

NIGHTLY

Valgrind + long stress

Release binaries under Memcheck; longer TSan soak runs.

⚠ **Rule: any sanitizer report fails the build**. No "warnings-only" mode, ever.

REGRESSION TESTING

Turn Every Report Into a Test

REGRESSION TESTING

Turn Every Report Into a Test

01

Triage

Group duplicate reports by their SUMMARY line: error type, file:line, function

REGRESSION TESTING

Turn Every Report Into a Test

01

Triage

Group duplicate reports by their SUMMARY line: error type, file:line, function

02

Minimize

Smallest test that reproduces it

REGRESSION TESTING

Turn Every Report Into a Test

01

Triage

Group duplicate reports by their SUMMARY line: error type, file:line, function

02

Minimize

Smallest test that reproduces it

03

Fix the cause

Ownership, lifetime or sync, not the symptom

REGRESSION TESTING

Turn Every Report Into a Test

01

Triage

Group duplicate reports by their SUMMARY line: error type, file:line, function

02

Minimize

Smallest test that reproduces it

03

Fix the cause

Ownership, lifetime or sync, not the symptom

04

Lock it in

Test runs under the tool that caught it

REGRESSION TESTING

Turn Every Report Into a Test

01

Triage

Group duplicate reports by their SUMMARY line: error type, file:line, function

02

Minimize

Smallest test that reproduces it

03

Fix the cause

Ownership, lifetime or sync, not the symptom

04

Lock it in

Test runs under the tool that caught it

05

No suppressions

Fix it rather than add it to a suppression file

REGRESSION TESTING

Turn Every Report Into a Test

01

Triage

Group duplicate reports by their SUMMARY line: error type, file:line, function

02

Minimize

Smallest test that reproduces it

03

Fix the cause

Ownership, lifetime or sync, not the symptom

04

Lock it in

Test runs under the tool that caught it

05

No suppressions

Fix it rather than add it to a suppression file

A fix without a sanitized regression test is a bug that will come back.

WHEN YOU CAN'T FIX IT

Managing Noise Without Losing Signal

valgrind.supp

```
# [TICKET-123] owner: [name]
# one-time init in vendor lib
{
   vendor_ssl_init
   Memcheck:Leak
   match-leak-kinds: reachable
   fun:malloc
   ...
   fun:OPENSSL_init_ssl
}

# generate candidates, then trim
$ valgrind --gen-suppressions=all ...
```

Suppress narrowly
One function or stack, never a whole library.

Owner & ticket
Every entry has someone responsible and a reason to exist.

”False positive” is a claim
Prove it: uninstrumented code? Custom sync? Usually it's a real bug.

BEYOND THE DESKTOP

Embedded: Host First, Then the Target

01

Host first

Build the target-independent code (protocols, parsers, state machines) for x86-64 and run its tests under ASan, UBSan and TSan.

02

Then the target

Run cross-built binaries under QEMU user-mode, or Valgrind directly on an ARM Linux board, with no rebuild.

On the device itself: **UBSan trap mode** or **HWASan**, both covered earlier.

SUMMARY

Key Takeaways

01 Green tests prove outputs, not memory safety, thread safety or defined behavior.

02 ASan + UBSan is the everyday build. TSan gets its own. Valgrind needs no rebuild.

03 Read the whole report: each stack answers a different question.

04 Fail CI on any report. A report is a bug, not a warning.

05 Every finding should become a regression test under the tool that caught it.

Q&A

Thank you

Alex Kushnir

MY E-CARD

Scan the QR code

SLIDES & EXAMPLES

github.com/alexkushnir/corecpp_2026_talk

qa.cpp — -fsanitize=all_of_them

```
while (audience.has_questions()) {
    answer(audience.next());
}

return Thanks{};

// ERROR SUMMARY: 0 errors
```