LLVM's Realtime Safety Revolution:

Tools for Modern Mission Critical Systems

DAVID TREVELYAN &
CHRISTOPHER APPLE

Chris Apple

- 10-year veteran of the audio industry
- Previously Dolby, Roblox, Spatial Inc.
- Currently: layabout

Authors of RealtimeSanitizer

David Trevelyan

- 15 years' experience in academia, startups and big tech
- Previously TikTok, Imperial College London
- Currently: audio software consultant

Special thanks

Alistair Barker
Co-author of RealtimeSanitizer

Doug Wyatt
Inventor/author of performance constraints

AGENDA

50%

1. Real-time programming
2. Existing strategies
3. RealtimeSanitizer
4. Performance constraints

50%

5. Comparing and contrasting

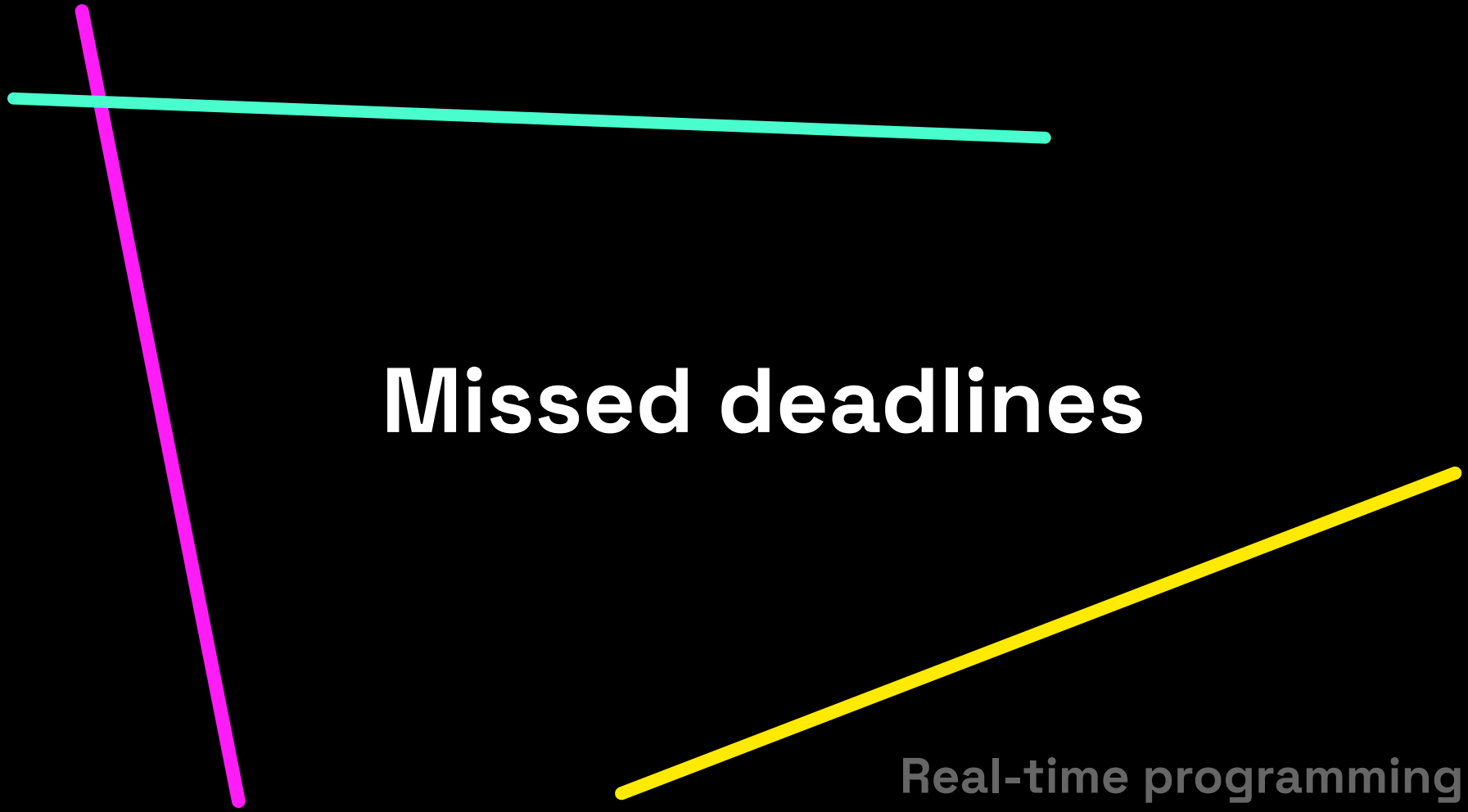
AGENDA

1. **Real-time programming**
2. Existing strategies
3. RealtimeSanitizer
4. Performance constraints
5. Comparing and contrasting

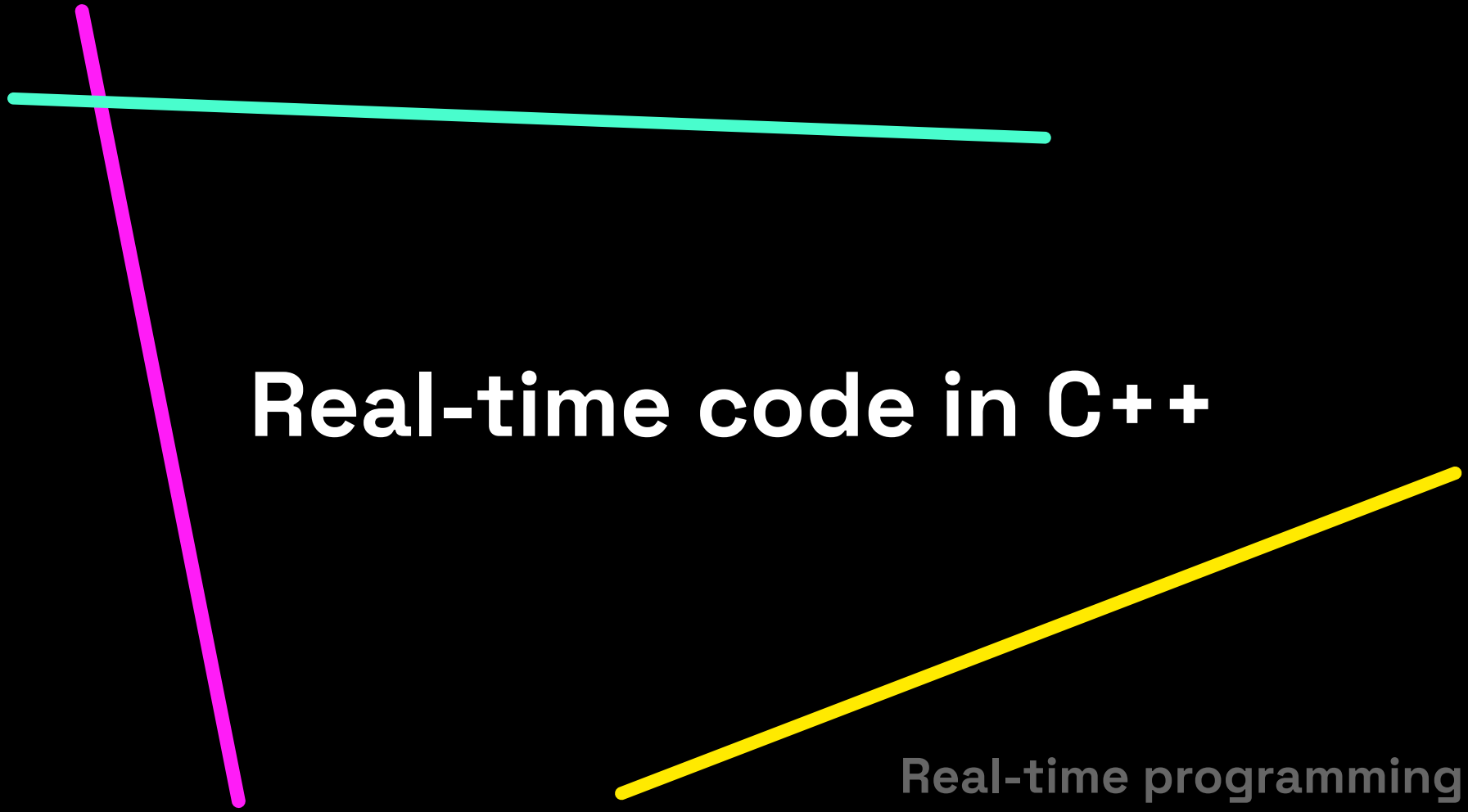
Real-time programs must **guarantee response** within **specified time constraints**, often referred to as **“deadlines”**.

1. Provide the right answer...

2. ... in the right time.


Missed deadlines


POP!


Real-time code in C++

```
void process (float * audio)
{
    // You have 11.6 ms from now, GO!!!!
    ...
}
```

```
void dispatch (Data& audio)
{
    // Fly a rocket...
    ...
}
```

Worst case execution time must be...

1. deterministic,
2. known in advance,
3. independent of input data, and
4. shorter than the given deadline [1].

[1] Timur Doumler - Real Time Programming with the C++ Standard Library

How long is a malloc?

10 ns	1,000,000 ns
90%	< 1%

How long is a real-time callback?

Milliseconds Microseconds Nanoseconds

Nondeterministic execution time

1. System calls
2. Allocations
3. Mutex locks/unlocks
4. Thrown exceptions
5. Indefinite waits (CAS loops, infinite loops)
6. …

[PROHIBITED]

nonblocking


What do violations
look like?

Sometimes it’s
obvious

```
void dispatch()
{
    mutex_.lock();
    ...
}
```

```
void dispatch()
{
    x = make_unique<Foo>();
    ...
}
```

Sometimes it’s

hidden

```
void process_audio()
{
    numbers_[1] = 2;
    ...
}
```

```
void dispatch()
{
    auto const x = input_array();
    auto const y = output_array();

    post_report([x, y](auto & data) {
        data.input = x;
        data.output = y;
     });
}
```

```
void process_audio()
{
    fftw_execute(plan);
    ...
}
```

How can we be confident that our code is real-time safe?

AGENDA

1. Real-time programming
2. **Existing strategies**
3. RealtimeSanitizer
4. Performance constraints
5. Comparing and contrasting

Existing strategies

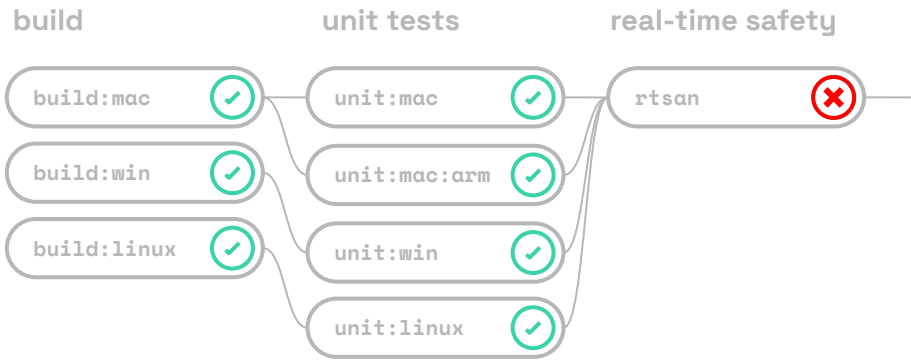
- Shared experience
- Code review
- Profilers and debuggers
- static_assert
- Documentation

- Getting experience takes a **long time**
- Code review is prone to **human error**
- Profiling/debugging is a **manual process**
- Static assertions are **limited**
- Documentation **goes out of date**
- *What about **pre-built dependencies**?*

What if we had a tool
that could
simply tell us?

A nice tool would...

- Assess real-time safety
- Detect a wide range of violations
- ...even from third-party and pre-compiled dependencies
- Point to any problematic code
- Be able to fail a CI pipeline



LLVM 20

1. RealtimeSanitizer
2. Performance constraints

AGENDA

1. Real-time programming
2. Existing strategies
3. **RealtimeSanitizer**
4. Performance constraints
5. Comparing and contrasting

Using sanitizers

```cpp
#include <vector>

int main()
{
    auto v = std::vector<int> (16);
    return v[16];
}
```

```
> clang -fsanitize=address main.cpp
> ./a.out
==98481==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x000105701320 at pc 0x000102d8
READ of size 4 at 0x000105701320 thread T0
   #0 0x5770e099d6c0 in main /app/example.cpp:6:12
   #1 0x7dbed6c29d8f  (/lib/x86_64-linux-gnu/libc.so.6+0x29d8f)
   #2 0x7dbed6c29e3f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x29e3f)
   #3 0x5770e08ba394 in _start (/app/output.s+0x2c394)

0x606000000300 is located 0 bytes after 64-byte region [0x6060000002c0,0x606000000300)
allocated by thread T0 here:
    #0 0x104c7f954 in _Znwm asan_new_delete.cpp:120
    #1 0x1044294cc in void* std::__1::__libcpp_operator_new[abi:v160006]<unsigned long>(unsigned
    #2 0x104429450 in std::__1::__libcpp_allocate[abi:v160006](unsigned long, unsigned long)+0x44
```

Using RealtimeSanitizer (RTSan)

```
float process(float x)  [[clang::nonblocking]]
{
    auto const y = std::vector<float> (16);
    ...
}
```

```
> clang++ -fsanitize=realtime main.cpp
> ./a.out
==86660==ERROR: RealtimeSanitizer: unsafe-library-call
Intercepted call to real-time unsafe function `malloc` in real-time context!
    #0 0x103184cfc in malloc rtsan_interceptors.cpp:225
    #1 0x18cb16524 in operator new(unsigned long)+0x1c
     ...
    #10 0x102c02b8c in std::__1::vector<float, std::__1::allocator<float>>::vector
    #11 0x102c02b38 in process(float)+0x28 /app/example.cpp:6:14
    #12 0x102c02c00 in main+0x1c /app/example.cpp:12:5
```

Two steps

1. Attribute real-time functions with **`[[clang::nonblocking]]`**
2. Compile and link with **`-fsanitize=realtime`**

How RealtimeSanitizer works

Tracking and interception

lightweight runtime library

```
void __rtsan_realtime_enter() { ... }
void __rtsan_realtime_exit() { ... }


INTERCEPTOR (void *, malloc, size_t size)
{
    if (is_in_realtime_context()):
       print_stack_and_die("malloc");

    return REAL(malloc)(size);
}
```

LLVM Intermediate Representation (IR)

Real-time context signaling

compilation step

```
int dispatch() {[clang::nonblocking]] {
    return calculate_result();
}
```

↓ -fsanitize=realtime

```
define noundef i32 @_Z8dispatchv() #1 {
  call void @__rtsan_realtime_enter()
  %1 = call noundef i32 @_Z16calculate_resultv()
  call void @__rtsan_realtime_exit()
  ret i32 %1
}
```

Interception

Interception allows us to:

- detect any function call
- assert not in real-time context
- error or behave normally

RealtimeSanitizer aims to intercept anything that could block

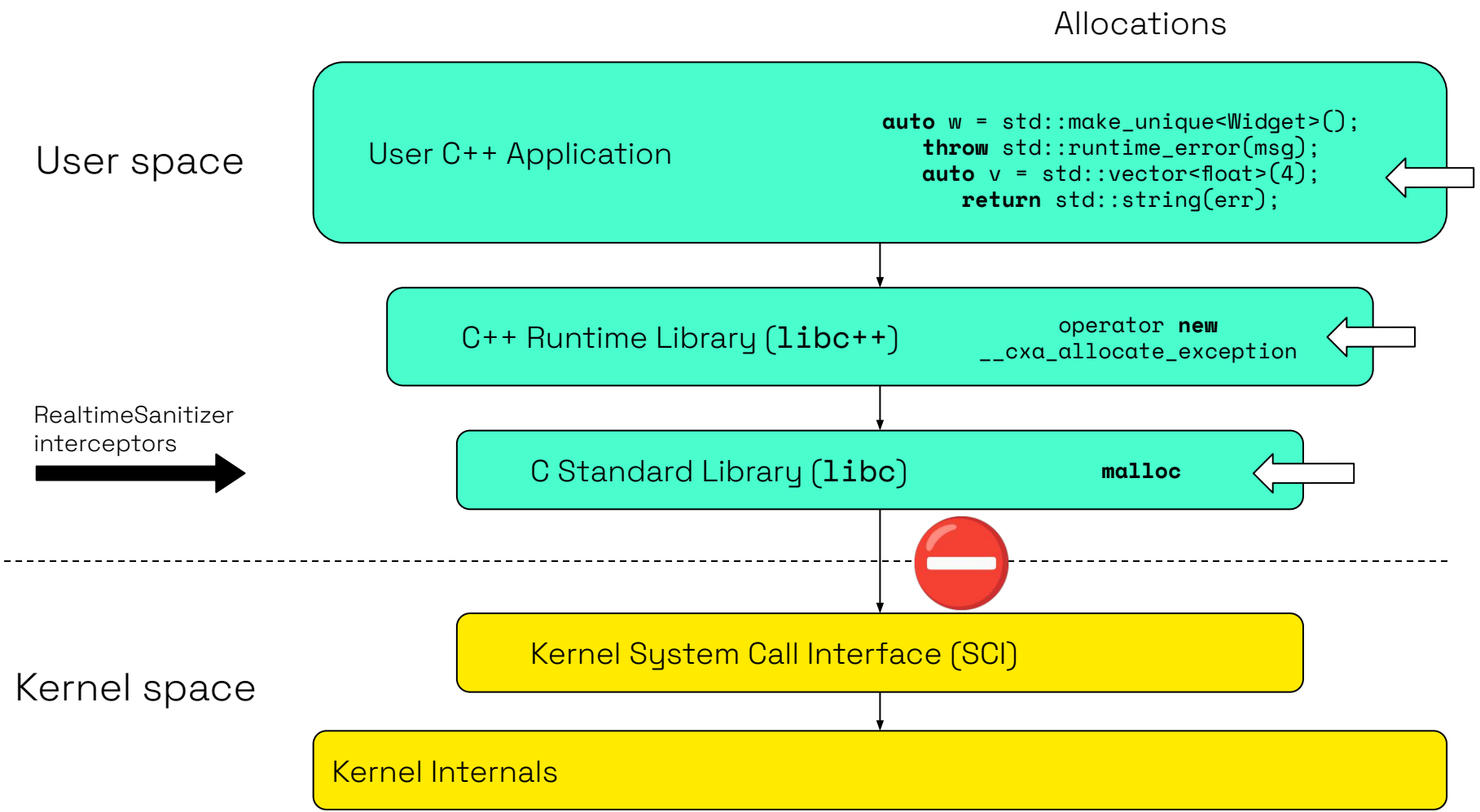
```
INTERCEPTOR (void *, malloc, size_t size) {
    __rtsan_expect_not_realtime("malloc");
    return REAL(malloc)(size);
}

INTERCEPTOR (void, free, void * ptr) {
    __rtsan_expect_not_realtime("free");
    return REAL(free)(ptr);
}

INTERCEPTOR (int, pthread_mutex_lock, pthread_mut
    __rtsan_expect_not_realtime("pthread_mutex_lo
    return REAL(pthread_mutex_lock)(mutex);
}


INTERCEPTOR (int, pthread_mutex_unlock, pthread_m
    __rtsan_expect_not_realtime("pthread_mutex_un
    return REAL(pthread_mutex_unlock)(mutex);
}

INTERCEPTOR (int, pthread_cond_signal, pthread_co
    __rtsan_expect_not_realtime("pthread_cond_sig
```


Allocations
User space
User C++ Application
auto w = std::make_unique<Widget>();
throw std::runtime_error(msg);
auto v = std::vector<float>(4);
return std::string(err);
C++ Runtime Library (libc++)
operator new
__cxa_allocate_exception
RealtimeSanitizer
interceptors
C Standard Library (libc)
malloc
Kernel space
Kernel System Call Interface (SCI)
Kernel Internals

Memory allocation

malloc, calloc

realloc, reallocf,

valloc, alligned_alloc

free

posix_memalign

Threads & sleep

pthread_create, pthread_join

pthread_mutex_lock, pthread_mutex_unlock

pthread_cond_signal, pthread_cond_broadcast

OSSpinLockLock, os_unfair_lock_lock

sleep, usleep, nanosleep

Filesystem & streams

open, openat, creat, close,

fopen, fopenat, fclose, fread, fwrite,

puts, fputs, read, write, writev, readv,

pwrite, pread

Sockets

socket

send, sendto, sendmsg

recv, recvfrom, recvmsg

shutdown

```cpp
float process(float x)  [[clang::nonblocking]]
{
    auto y = std::vector<float> (16);
    ...
}
```

```
> clang++ -fsanitize=realtime main.cpp

> ./a.out
==86660==ERROR: RealtimeSanitizer: unsafe-library-call
Intercepted call to real-time unsafe function `malloc` in real-time context!
    #0 0x103184cfc in malloc rtsan_interceptors.cpp:225
    #1 0x18cb16524 in operator new(unsigned long)+0x1c
     ...
    #10 0x102c02b8c in std::__1::vector<float, std::__1::allocator<float>>
    #11 0x102c02b38 in process(float)+0x28 /app/example.cpp:6:14
    #12 0x102c02c00 in main+0x1c /app/example.cpp:12:5
```

AGENDA

1. Real-time programming
2. Existing strategies
3. RealtimeSanitizer
4. **Performance constraints**
5. Comparing and contrasting

Performance constraint compilation flags

```
-Wfunction-effects

-Wperf-constraint-implies-noexcept
```

Performance constraint function attributes

```
[[clang::nonallocating]]

[[clang::nonblocking]]
```

[[clang::nonallocating]]

1. Add `[[clang::nonallocating]]` to a function.
2. Compile with warning flags

   `-Wfunction-effects`

   `-Wperf-constraint-implies-noexcept`

```
void process()
[[clang::nonallocating]] {

   float* v = new float[100];

}
```

```
warning: 'nonallocating' function must not
allocate or deallocate memory
[-Wfunction-effects]
   16 |   float* v = new float[100];
```

[[clang::nonblocking]]

1. Add `[[clang::nonblocking]]` to a function.
2. Compile with warning flags

   `-Wfunction-effects`

   `-Wperf-constraint-implies-noexcept`

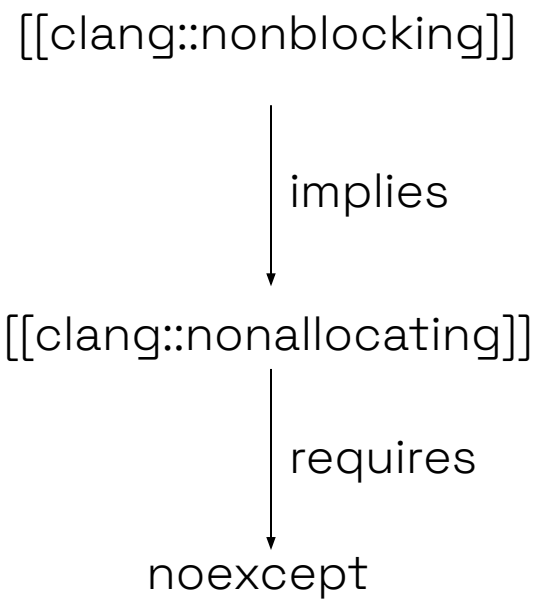
```
void dispatch()
[[clang::nonblocking]] {

   m.lock();

}
```

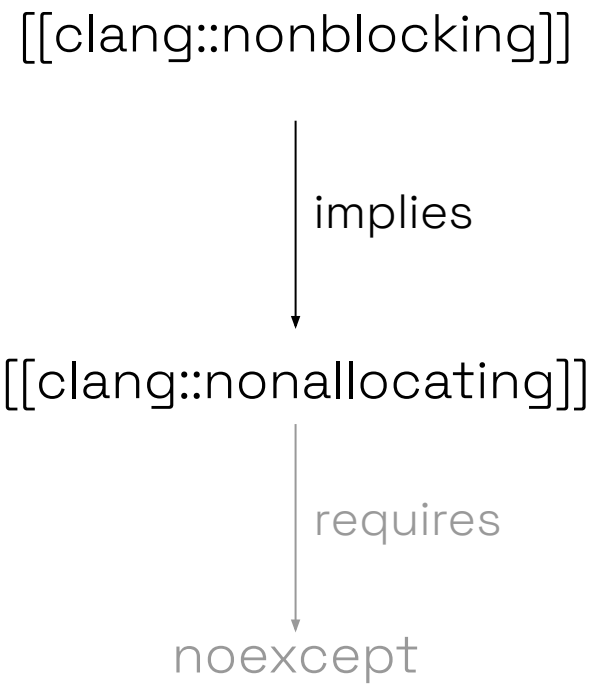
```
warning: 'nonblocking' function must not call
non-'nonblocking' function 'std::mutex::lock'
[-Wfunction-effects]
     5 |   m.lock();
```

Performance constraints attributes



```
void dispatch() noexcept
[[clang::nonblocking]] {
 float* ptr = new float;
}
```

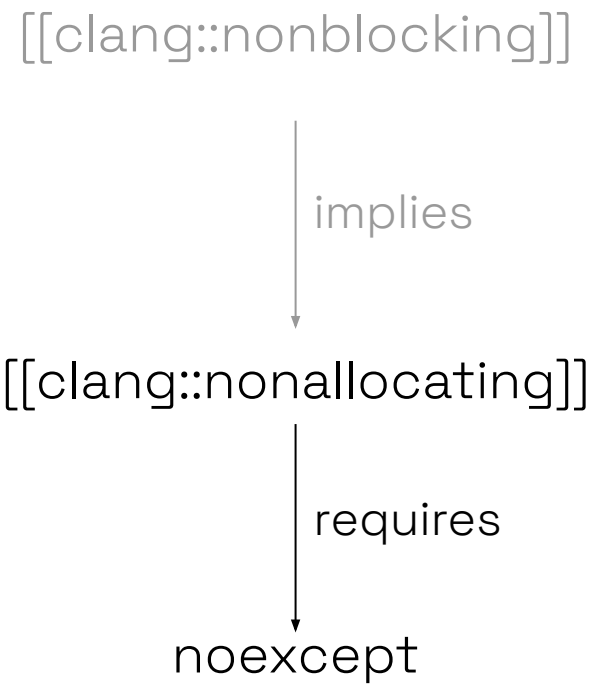
```
warning: 'nonblocking' function must
not allocate or deallocate memory
[-Wfunction-effects]
    4 |   float* ptr = new float;
```



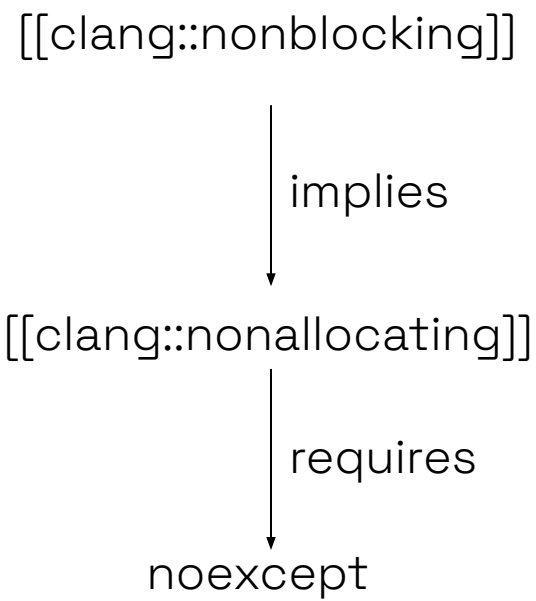
```
void foo()
[[clang::nonallocating]];
```

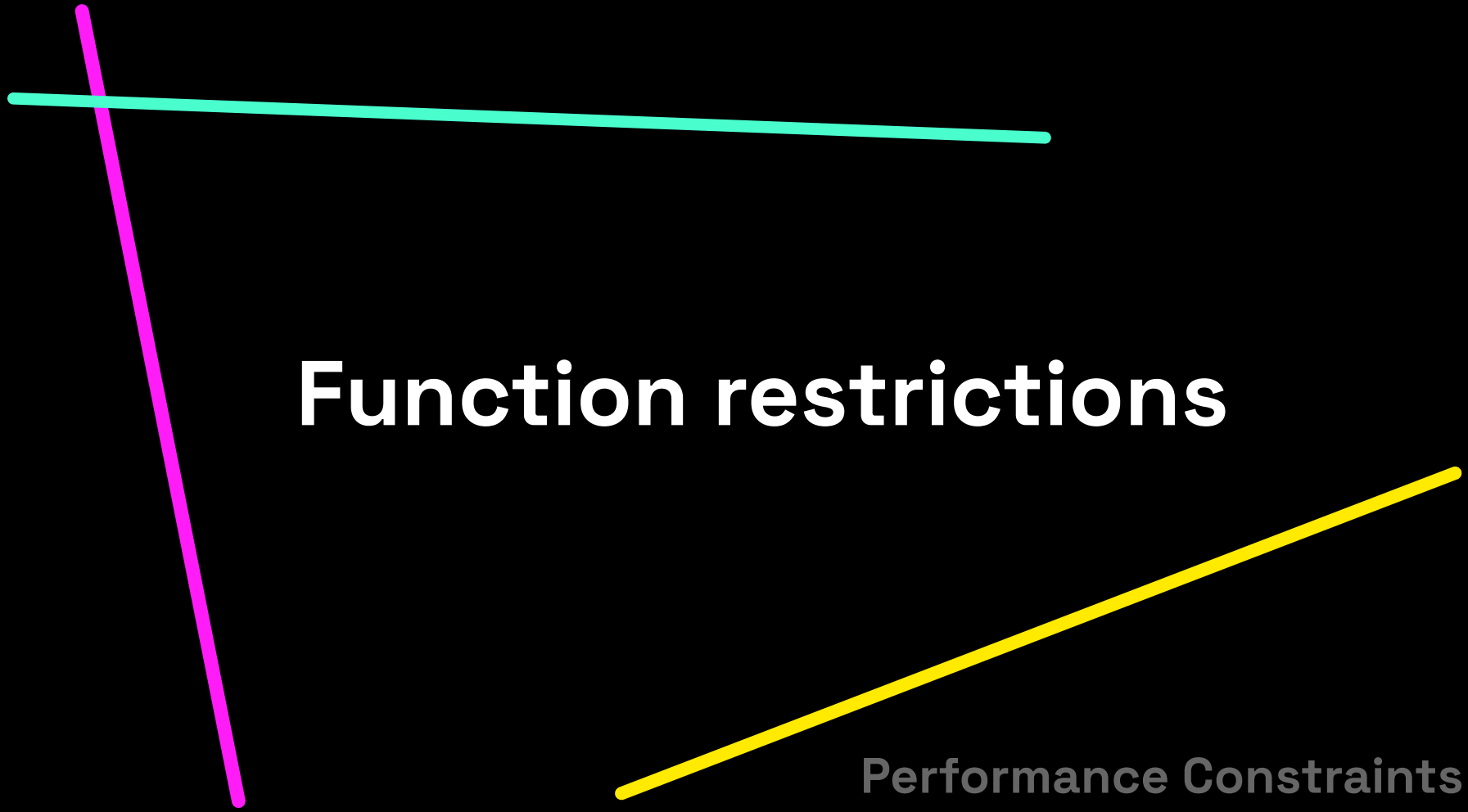
```
warning: 'nonallocating'
function should be declared
noexcept
    3 | void foo()
```

-Wperf-constraint-implies-noexcept



Performance constraints attributes




Function restrictions

Normal functions can call...

[[nonblocking]]

[[nonallocating]]

no attributes

[[nonallocating]] can call...

[[nonblocking]]

[[nonallocating]]

~~no attributes~~

[[nonblocking]] can call...

[[nonblocking]]

~~[[nonallocating]]~~

~~no attributes~~

Function call restrictions

```
void unknown_function() [[clang::nonblocking]];

void process() noexcept [[clang::nonblocking]] {
 unknown_function();
}
```

Missing [[nonblocking]]

```
warning: 'nonblocking' function must not call non-'nonblocking'
function 'unknown_function' [-Wfunction-effects]
    7 |  unknown_function();
```

[[nonblocking]] can ONLY call other functions marked [[nonblocking]]

Functions may only call their constraint, or stricter

[[clang::nonblocking]]

[[clang::nonallocating]]

Does it compile?

```
void foo() [[clang::nonblocking]];


void process() [[clang::nonblocking]]
{
    foo();
}
```

✅

[[clang::nonblocking]]

[[clang::nonallocating]]

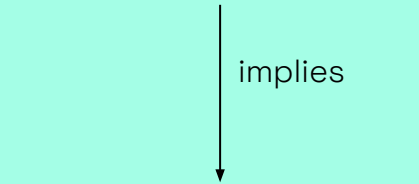
Does it compile?

```
void foo();


void process() [[clang::nonblocking]]
{
    foo();
}
```

```
main.cpp:6:3: warning: 'nonblocking'
function must not call non-'nonblocking'
function 'foo' [-Wfunction-effects]
    6 |   foo();
```

[[clang::nonblocking]]



[[clang::nonallocating]]

Does it compile?

```
void foo() [[clang::nonallocating]];


void process() [[clang::nonblocking]]
{
    foo();
}
```

```
main.cpp:6:3: warning: 'nonblocking' function
must not call non-'nonblocking' function
'foo' [-Wfunction-effects]
    6 |   foo();
```

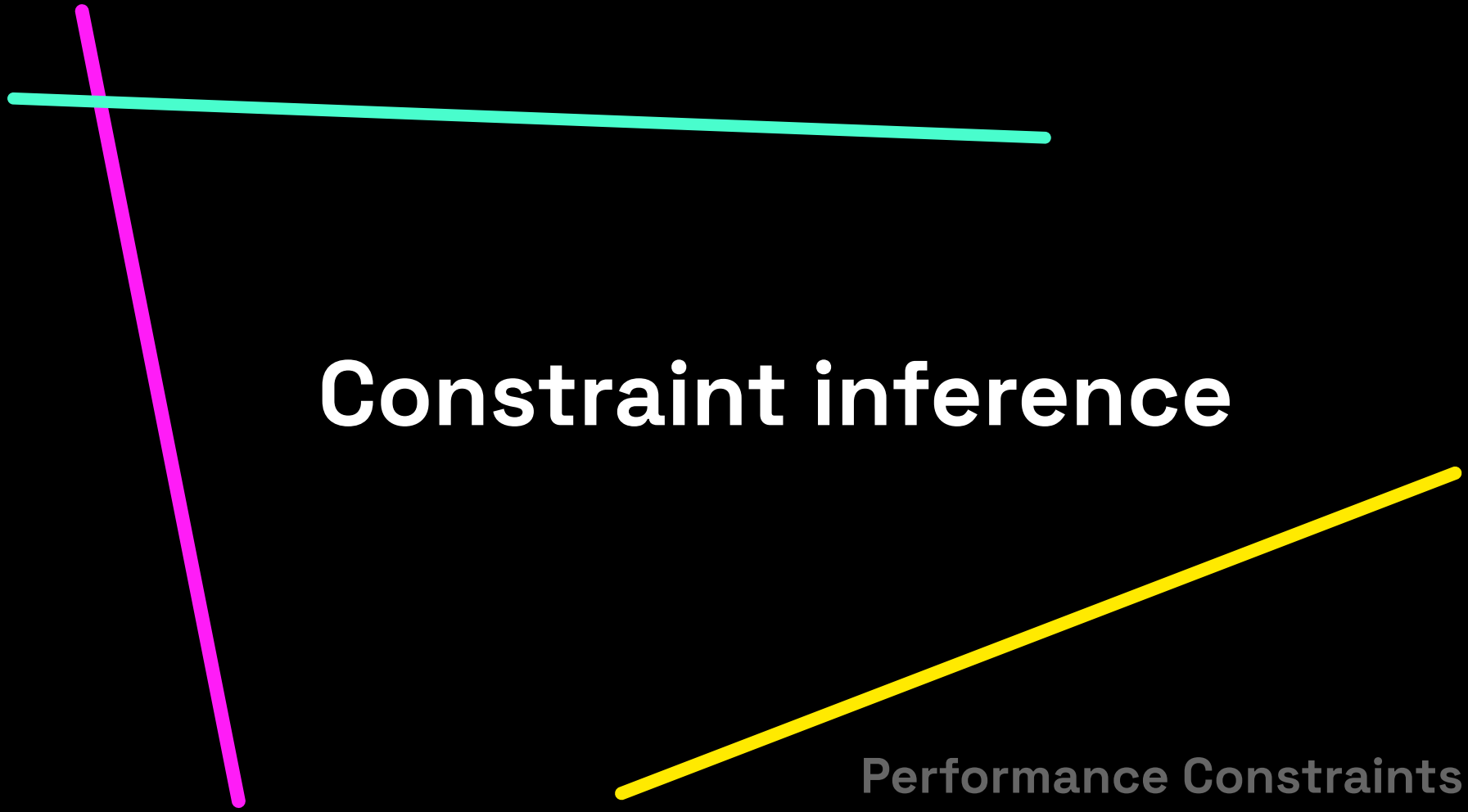
[[clang::nonblocking]]

[[clang::nonallocating]]

Does it compile?

```
void foo() [[clang::nonblocking]];


void process() [[clang::nonallocating]]
{
    foo();
}
```


Constraint inference

Inference - Same TU

```
int defined_here() {}



void process() noexcept
[[clang::nonblocking]] {
  defined_here();
}
```

```
int defined_here() {
  mutex.lock();
}

void process() noexcept
[[clang::nonblocking]] {
  defined_here();
}
```

Inference - Same TU

```
main.cpp:9:3: warning:
'nonblocking' function must not
call non-'nonblocking' function
'defined_here' [-Wfunction-effects]
    9 |   defined_here();

main.cpp:17:3: note: function
cannot be inferred 'nonblocking'
because it calls non-'nonblocking'
function 'std::mutex::lock'
   17 |   mutex.lock();
```

```
// third_party.h
void defined_elsewhere();


// main.cpp


void process() noexcept
[[clang::nonblocking]] {
  defined_elsewhere();
}
```

Inference - Different TU

```
main.cpp:5:3: warning: 'nonblocking'
function must not call non-'nonblocking'
function 'defined_elsewhere'
[-Wfunction-effects]

    5 |   defined_elsewhere();
```

```
main.cpp:2:5: note: declaration cannot be
inferred 'nonblocking' because it has no
definition in this translation unit

    2 | int defined_elsewhere();
```

Re-declaration

```
// third_party.h
void defined_elsewhere();

// main.cpp
void defined_elsewhere() [[clang::nonblocking]];
void process() [[clang::nonblocking]] {
  defined_elsewhere();
}
```

Re-declaration

```
// third_party.h
void defined_elsewhere();

// main.cpp
void defined_elsewhere() [[clang::nonblocking]];
void process() [[clang::nonblocking]] {
  defined_elsewhere();
}
```

```
// elsewhere.cpp
void defined_elsewhere()
{
    mutex.lock();
    …
}
```

This will compile without warnings, even though it is incorrect!

Summary of performance constraints

```
 > clang++ -Wfunction-effects -Wperf-constraint-implies-noexcept main.cpp
void process() noexcept [[clang::nonblocking]] {
  float* f = new float;
  // error: 'nonblocking' function must not allocate or deallocate memory

  auto g = foo();
  // error: 'nonblocking' function must not call non-'nonblocking' function 'foo'

  static int x = 0;
  // error: 'nonblocking' function must not have static locals
}
```

AGENDA

1. Real-time programming
2. Existing strategies
3. RealtimeSanitizer
4. Performance constraints
5. **Comparing and contrasting**

INTERLUDE

WARNING!

Neither RTSan nor the perf. constraints attributes can fully guarantee real-time safety

RTSan blind spots

- No guarantee of processor time.
- No guarantee your code runs faster than allotted time.
- No detection of hand-written assembly system calls.
- Not all libc wrapper functions implemented.
- No detection of nondeterministic loops.
  - Infinite loops
  - Nondeterministic loops (CAS)

RTSan blind spots

- No guarantee of processor time.
- No guarantee your code runs faster than allotted time.
- No detection of hand-written assembly system calls.
- Not all libc wrapper functions implemented.
- No detection of nondeterministic loops.
  - Infinite loops
  - Nondeterministic loops (CAS)

Perf. constraints blind spots

- No guarantee of processor time.
- No guarantee your code runs faster than allotted time.
- Misdeclared functions.

Perf. constraints blind spots

- No guarantee of processor time.
- No guarantee your code runs faster than allotted time.
- Misdeclared functions.

Neither RTSan nor the perf. constraints attributes can fully guarantee real-time safety

These tools make writing real-time code safer

INTERLUDE
OVER

AGENDA

1. Real-time programming
2. Existing strategies
3. RealtimeSanitizer
4. Performance constraints
5. **Comparing and contrasting**

RealtimeSanitizer

Performance constraints

Both

Detect real-time safety issues

Use [[clang::nonblocking]]

No real-time safety guarantee

1. Run time vs compile time
2. False negatives and false positives
3. Cost
4. Disabling each tool
5. Using each tool outside of LLVM

1. **Run time vs compile time**
2. False negatives and false positives
3. Cost
4. Disabling each tool
5. Using each tool outside of LLVM

RealtimeSanitizer

Performance constraints

Run time

Compile time

RealtimeSanitizer

If your code runs, it is compliant.*

*** As long as you hit every path in your code.**

```
if (something_rare())
    return process_that_allocates();
```

Recommendation:

1. Extensive unit testing with RealtimeSanitizer
2. QA testing with RealtimeSanitizer

Performance constraints

If your code compiles without warnings, it is compliant.*

*** As long as you:**

1. **Pay attention to the warnings**
2. **Declare everything correctly, especially in third party dependencies**

Recommendation:

1. Compile with **`-Werror=function-effects`** and **`-Werror=perf-constraint-implies-noexcept`**
2. “Audit” 3rd party code with RealtimeSanitizer

RealtimeSanitizer

With RealtimeSanitizer enabled:

- **Extensive unit testing**
- **Extensive QA testing**

Performance constraints

Compile with -Werror

“Audit” 3rd party code with RealtimeSanitizer

1. **Run time vs compile time**
2. False negatives and false positives
3. Cost
4. Disabling each tool
5. Using each tool outside of LLVM

1. Run time vs compile time
2. **False negatives and false positives**
3. Cost
4. Disabling each tool
5. Using each tool outside of LLVM

RealtimeSanitizer

Possible false negatives

Performance constraints

Possible false positives

Perf. constraints

```
main.cpp:8:3: error: 'nonblocking'
function must not call non-'nonblocking'
function 'std::vector<int>::push_back'

    8 |   v.push_back(3);
```

push_back real-time safe?

```
void dispatch(vector<int>& v)
noexcept [[clang::nonblocking]]
{
  v.clear();
  v.push_back(3);
}
```

```
…

vector<int> v;

v.reserve(512);

dispatch(v);

…
```

push_back real-time safe?

RealtimeSanitizer

```
void dispatch(vector<int>& v)
noexcept [[clang::nonblocking]]
{
  v.clear();
  v.push_back(3);
}
```

…

```
vector<int> v;
v.reserve(512);
dispatch(v);
```

…

Pre-reserved push_back

RTSan

- Did not error
- “This method is compliant on all the paths we hit!”

Performance constraints

- False positive, showed an error when the code did not allocate.
- “This method is not always free of non-blocking calls!”

Especially useful for probing third-party libraries and the STL!

Perf. constraints analysis has no false negatives

But it is prone to false positives

RealtimeSanitizer has no false positives

But it is prone to false negatives

1. Run time vs compile time
2. **False negatives and false positives**
3. Cost
4. Disabling each tool
5. Using each tool outside of LLVM

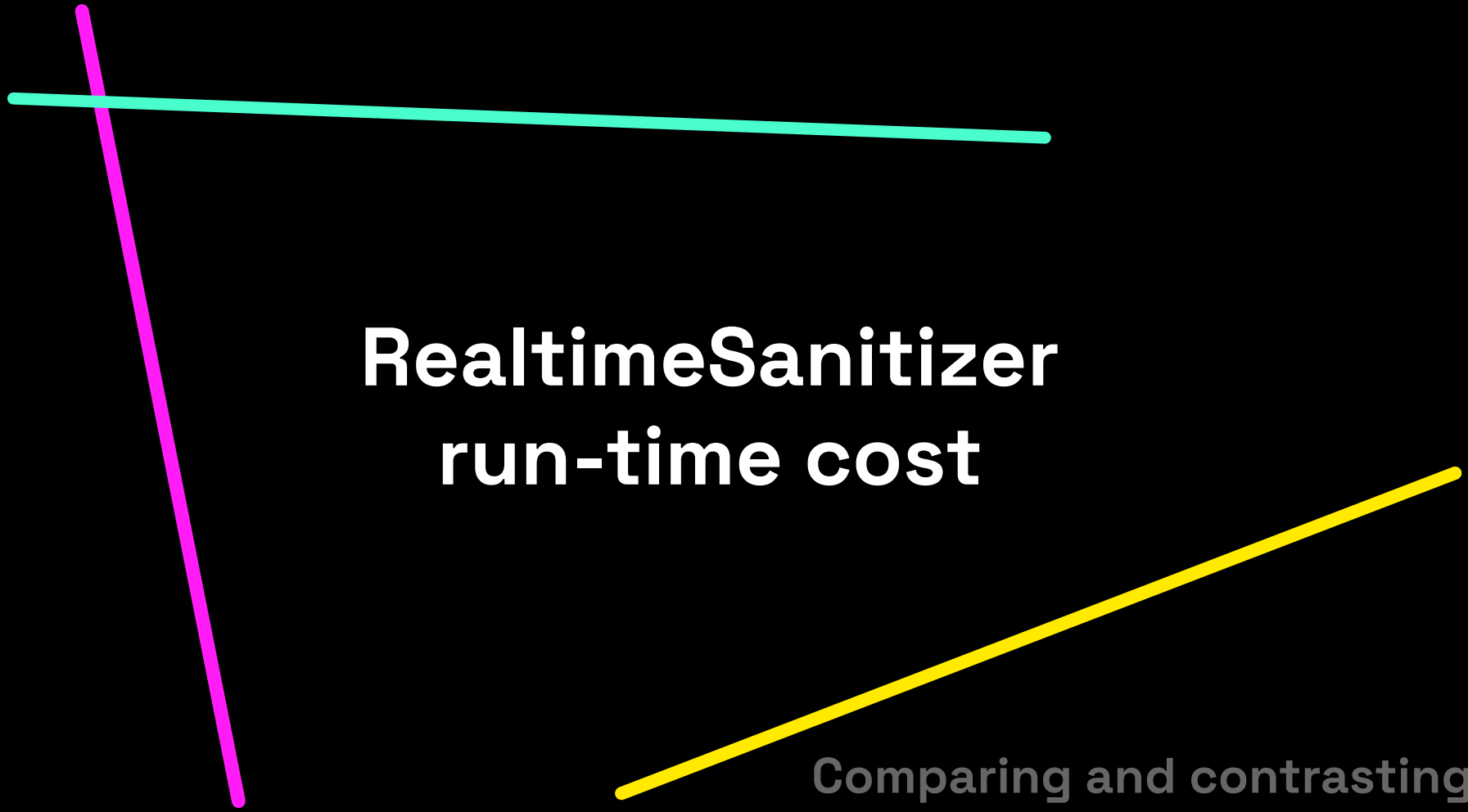
1. Run time vs compile time
2. False negatives and false positives
3. **Cost**
4. Disabling each tool
5. Using each tool outside of LLVM

RealtimeSanitizer

Performance constraints

Run time cost

Code conversion cost


RealtimeSanitizer
run-time cost
Comparing and contrasting

How does RTSan change your code?

```
int dispatch() [[clang::nonblocking]] {
    __rtsan_realtime_enter();
    ...fly a rocket code here...
    __rtsan_realtime_exit();
}
```

How does RTSan change your code?

```
int dispatch() [[clang::nonblocking]] {
    realtime_depth++; // thread-local integer
    ...fly a rocket code here...
    realtime_depth--;
}
```

`[[nonblocking]]`

- Every `[[nonblocking]]` function has an additional increment and decrement of an integer

- May interfere with inlining optimizations

Non-constrained?

How does RTSan change your code?

```
int ui_thread() {


                auto w = std::make_unique<Widget>(...);

}
```

How does RTSan change your code?

```
int ui_thread() {


            Widget* w = (Widget*)malloc(sizeof(Widget));

}
```

How does RTSan change your code?

```
int ui_thread() {


                Widget* w = (Widget*)rtsan_malloc(sizeof(Widget));

}
```

How does RTSan change your code?

```
int ui_thread() {
    if (not __rtsan_in_dlsym())
       if (__rtsan_initialized())
          if (not __rtsan_in_realtime_context())
             Widget* w = (Widget*)real_malloc(sizeof(Widget));

}
```

`[[nonblocking]]`

- Every `[[nonblocking]]` function has an additional increment and decrement of an integer

- May interfere with inlining optimizations

Non-constrained

- A few additional function calls inserted between the intercepted call, and the real method being invoked
  - These functions are relatively light, checking state stored in integers

`[[nonblocking]]`

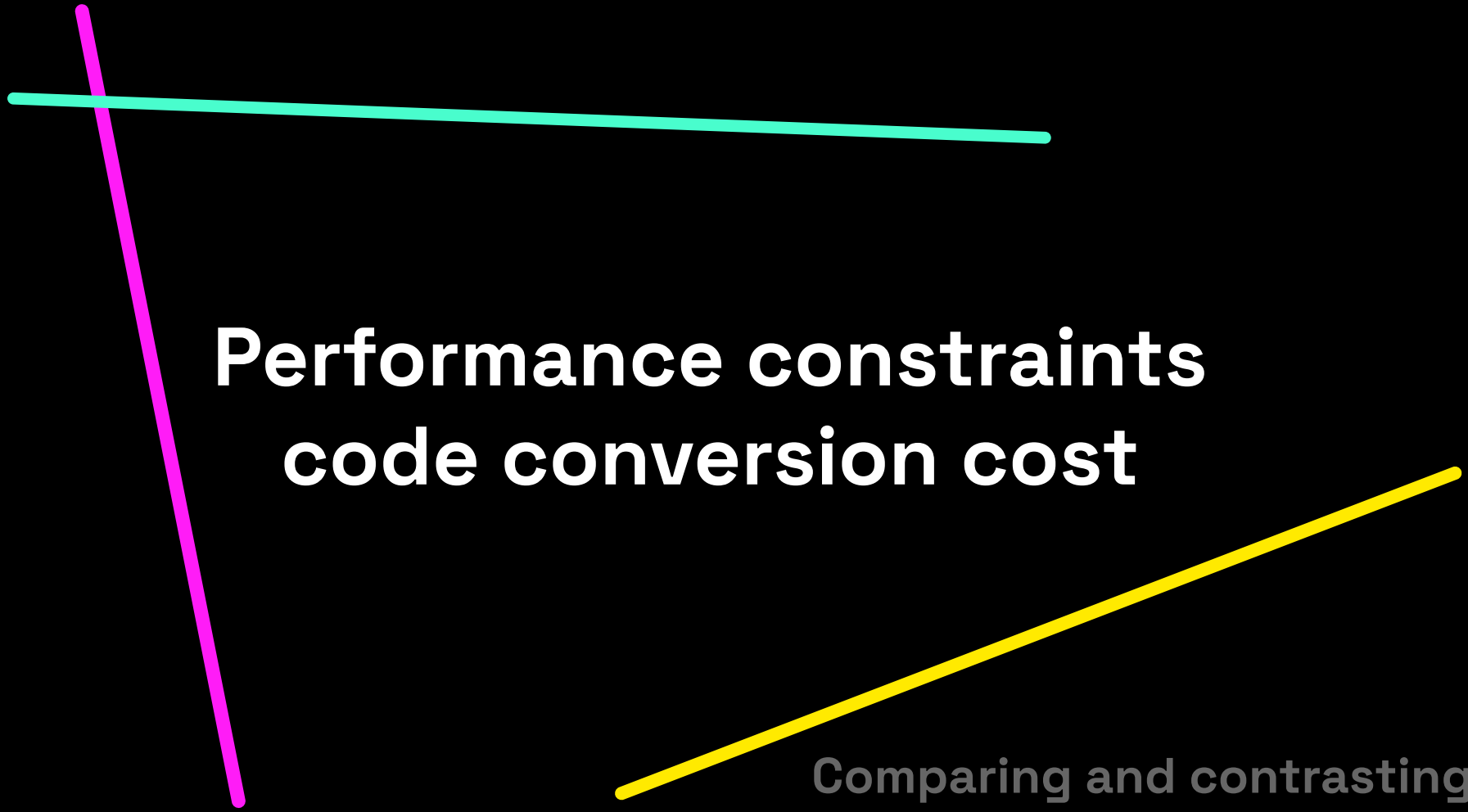
- Every `[[nonblocking]]` function has an additional increment and decrement of an integer
- May interfere with inlining optimizations

Non-constrained

- A few additional function calls inserted between the intercepted call, and the real method being invoked
  - These functions are relatively light, checking state stored in integers

***When the sanitizer is enabled*, there is a (minimal) runtime cost**

RealtimeSanitizer has no runtime cost without *-fsanitize=realtime*


Performance constraints
code conversion cost
Comparing and contrasting

Converting a codebase

```
void process() noex
[[clang::nonblockin
{
  foo();
}

 void foo() noexcept
 [[clang::nonblockin
 {
    bar();
 };
```

```
oexcept
blocking]]

locking' function
non-'nonblocking'
function
```

Upfront effort to convert to perf. constraints attributes

But it's *worth it*

No engineering cost for new projects

1. Run time vs compile time
2. False negatives and false positives
3. **Cost**
4. Disabling each tool
5. Using each tool outside of LLVM

1. Run time vs compile time
2. False negatives and false positives
3. Cost
4. **Disabling each tool**
5. Using each tool outside of LLVM

RealtimeSanitizer

Disable with `__rtsan::ScopedDisabler`

Performance constraints

Disable using call-site macro

Disabling function effects warnings

```
void process() noexcept
[[clang::nonblocking]]
{
  foo();
}
```

Disabling function effects warnings

```
void process() noexcept
[[clang::nonblocking]]
{
  NONBLOCKING_UNSAFE(foo());
}
```

```
#define NONBLOCKING_UNSAFE(...)                                    \
  _Pragma("clang diagnostic push")                                 \
  _Pragma("clang diagnostic ignored \"-Wunknown-warning-option\"") \
  _Pragma("clang diagnostic ignored \"-Wfunction-effects\"")       \
  __VA_ARGS__                                                      \
  _Pragma("clang diagnostic pop")
```

Disabling RTSan

```
#include <sanitizer/rtsan_interface.h>

void lock_error_mutex(std::mutex& m)
{
  __rtsan::ScopedDisabler disabler{};
  m.lock();
}

void process()
  noexcept [[clang::nonblocking]]
{
  if (buffer_overflow)
    lock_error_mutex(m);
}
```

“Real-time Confessions in C++” - Fabian Renn-Giles - ADC23

1. Run time vs compile time
2. False negatives and false positives
3. Cost
4. **Disabling each tool**
5. Using each tool outside of LLVM

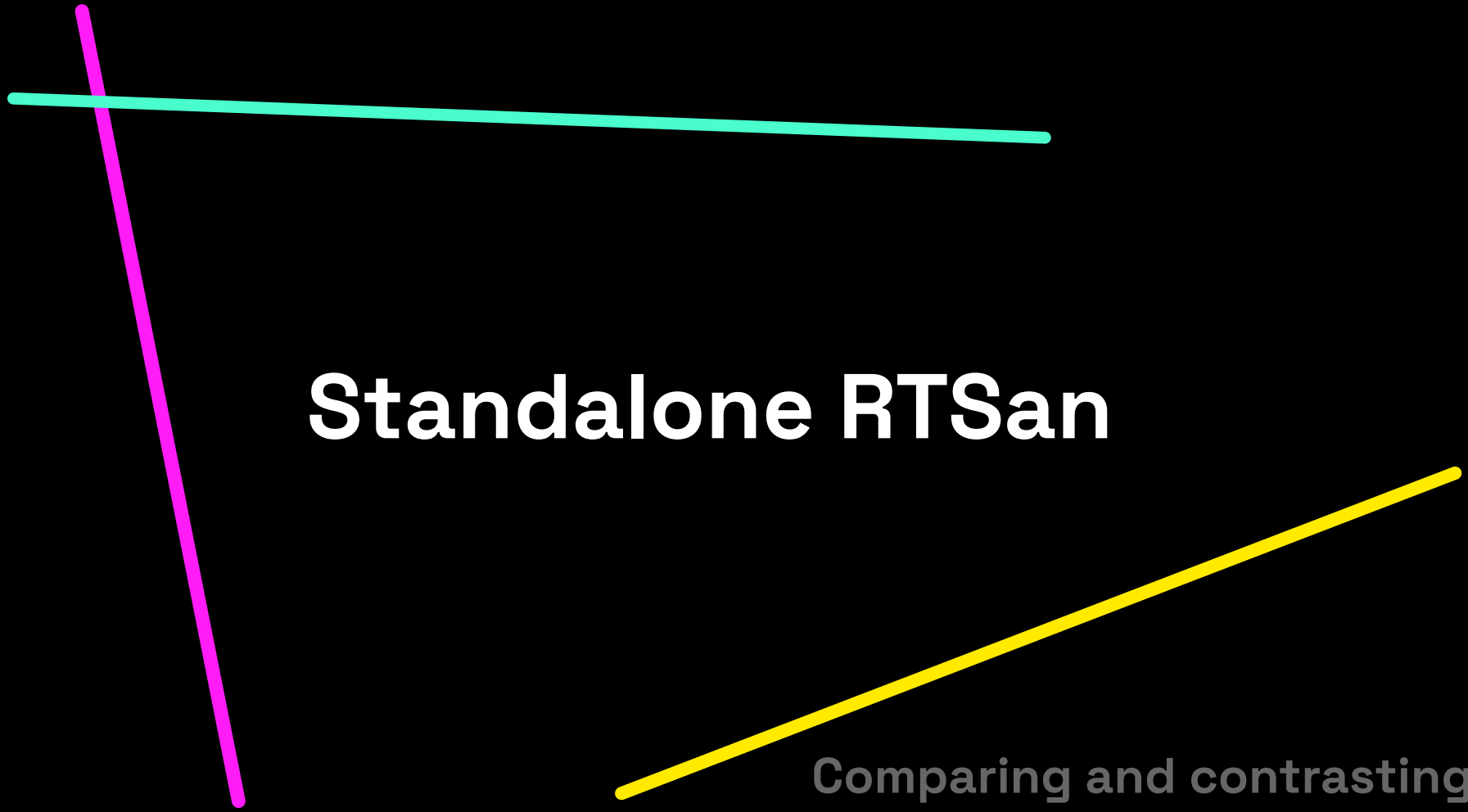
1. Run time vs compile time
2. False negatives and false positives
3. Cost
4. Disabling each tool
5. **Using each tool outside of LLVM**

RealtimeSanitizer

Can be used with other compilers (with a little hack)

Performance constraints

Dependent on `clang 20`


Standalone RTSan
Comparing and contrasting

realtime-sanitizer/rtsan

```
#define __SANITIZE_REALTIME
#include “rtsan_standalone/rtsan_standalone.h”
```

Standalone RTSan

1. `#define __SANITIZE_REALTIME` to conditionally enable the sanitizer
2. Find and include `rtsan_standalone.h`

Standalone RTSan

```
#define __SANITIZE_REALTIME
#include “rtsan_standalone/rtsan_standalone.h”

void main() {
  __rtsan::Initialize();
  ...
}

void process() {
  __rtsan::ScopedSanitizeRealtime ssr;

  auto x = std::make_unique<float>(3.0f);
}
```

1. `#define __SANITIZE_REALTIME` to conditionally enable the sanitizer
2. Find and include `rtsan_standalone.h`
3. Initialize rtsan early in your process
4. Insert ScopedSanitizeRealtime in any blocks that would be `[[nonblocking]]`

Standalone RTSan

```
#define __SANITIZE_REALTIME
#include “rtsan_standalone/rtsan_standalone.h”

void main() {
  __rtsan::Initialize();
  ...
}

void process() {
  __rtsan::ScopedSanitizeRealtime ssr;

  auto x = std::make_unique<float>(3.0f);
}


// CMakeLists.txt

target_link_libraries(helloWorld
  PRIVATE libclang_rt.rtsan_osx_dynamic.dylib
)
```

1. `#define __SANITIZE_REALTIME` to conditionally enable the sanitizer
2. Find and include `rtsan_standalone.h`
3. Initialize rtsan early in your process
4. Insert ScopedSanitizeRealtime in any blocks that would be `[[nonblocking]]`
5. Build and link the rtsan runtime from LLVM

Standalone RTSan

```
#define __SANITIZE_REALTIME
#include “rtsan_standalone/rtsan_standalone.h”

void main() {
  __rtsan::Initialize();
  ...
}

void process() {
  __rtsan::ScopedSanitizeRealtime ssr;

  auto x = std::make_unique<float>(3.0f);
}


// CMakeLists.txt

target_link_libraries(helloWorld
  PRIVATE libclang_rt.rtsan_osx_dynamic.dylib
)
```

1. `#define __SANITIZE_REALTIME` to conditionally enable the sanitizer
2. Find and include `rtsan_standalone.h`
3. Initialize rtsan early in your process
4. Insert ScopedSanitizeRealtime in any blocks that would be `[[nonblocking]]`
5. Build and link the rtsan runtime from LLVM

Standalone RTSan

AppleClang 15 ✅

GNU 14 on Linux ✅

realtime-sanitizer/rtsan

Summary

How do I use performance constraints attributes?

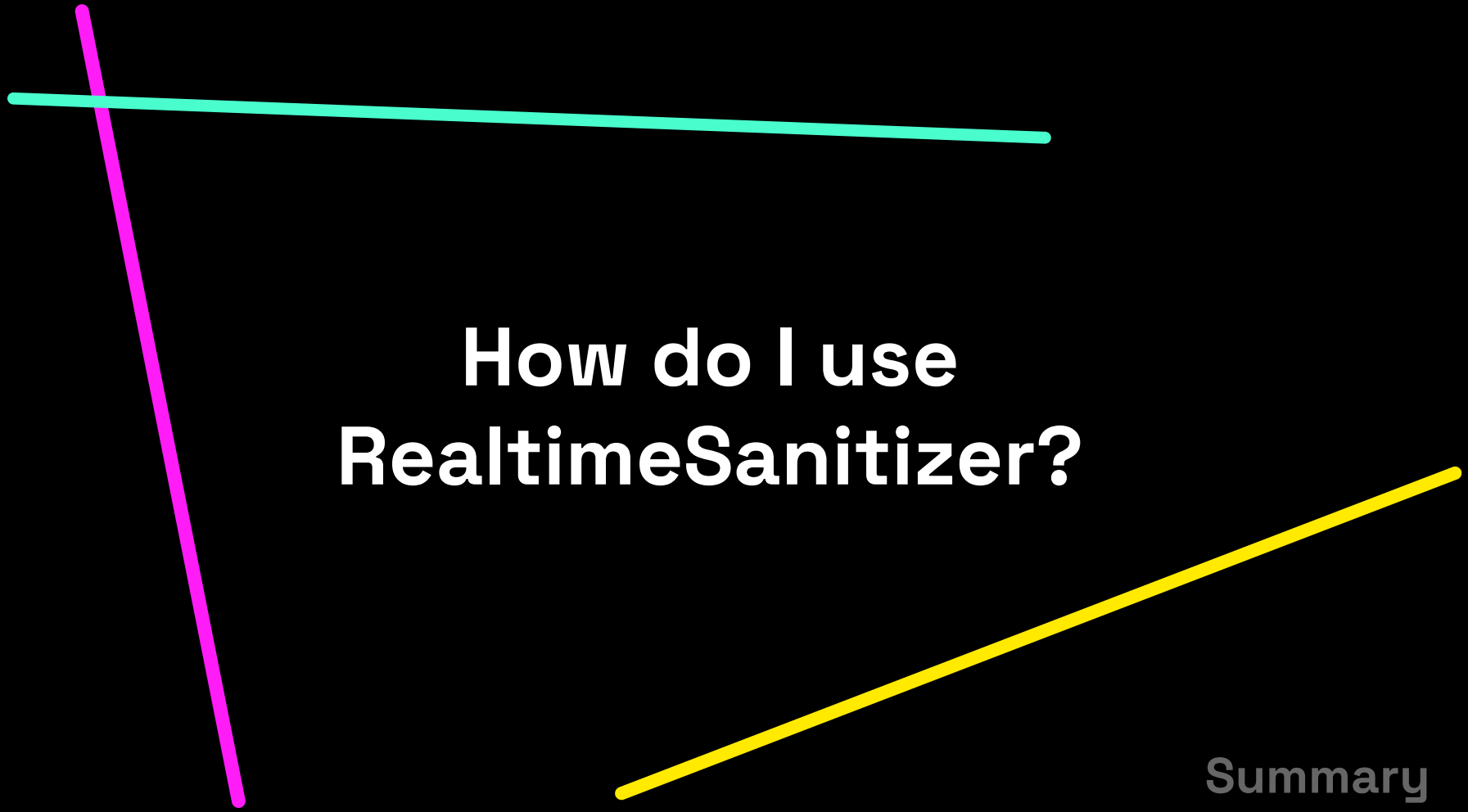
Summary

Summary: performance constraints attributes

```
float process (float x) noexcept [[clang::nonblocking]]
{
    auto y = std::vector<float> (16);
    ...
}
```

```
> clang++ -Wfunction-effects -Wperf-constraint-implies-noexcept main.cpp

main.cpp:22:8: error: 'nonblocking' function must not call non-'nonblocking' function
'std::vector<float>::~vector' [-Werror,-Wfunction-effects]
   22 |   auto y = std::vector<float> (16);
      |        ^
main.cpp:22:12: note: in template expansion here
   22 |   auto y = std::vector<float> (16);
      |        ^
main.cpp:22:12: error: 'nonblocking' function must not call non-'nonblocking' function
'std::vector<float>::vector' [-Werror,-Wfunction-effects]
   22 |   auto y = std::vector<float> (16);
      |        ^
```



Summary

Summary: RealtimeSanitizer

```
float process (float x) noexcept [[clang::nonblocking]]
{
    auto y = std::vector<float> (16);
    ...
}

> clang++ -fsanitize=realtime main.cpp


> ./a.out
==86660==ERROR: RealtimeSanitizer: unsafe-library-call
Intercepted call to real-time unsafe function `malloc` in real-time context!
    #0 0x103184cfc in malloc rtsan_interceptors.cpp:225
    #1 0x18cb16524 in operator new(unsigned long)+0x1c
    #2 0x102c03590 in std::__1::__libcpp_allocate
    ...
    #11 0x102c02b38 in process(float)+0x28 /app/example.cpp:6:14
    #12 0x102c02c00 in main+0x1c /app/example.cpp:12:5
```

RealtimeSanitizer

- Detect issues at run time
- Possible false negatives
- Run time cost
- Disable using ScopedDisabler
- Can be used with other compilers

Both

Detect real-time safety issues

Use [[clang::nonblocking]]

Perf. constraints

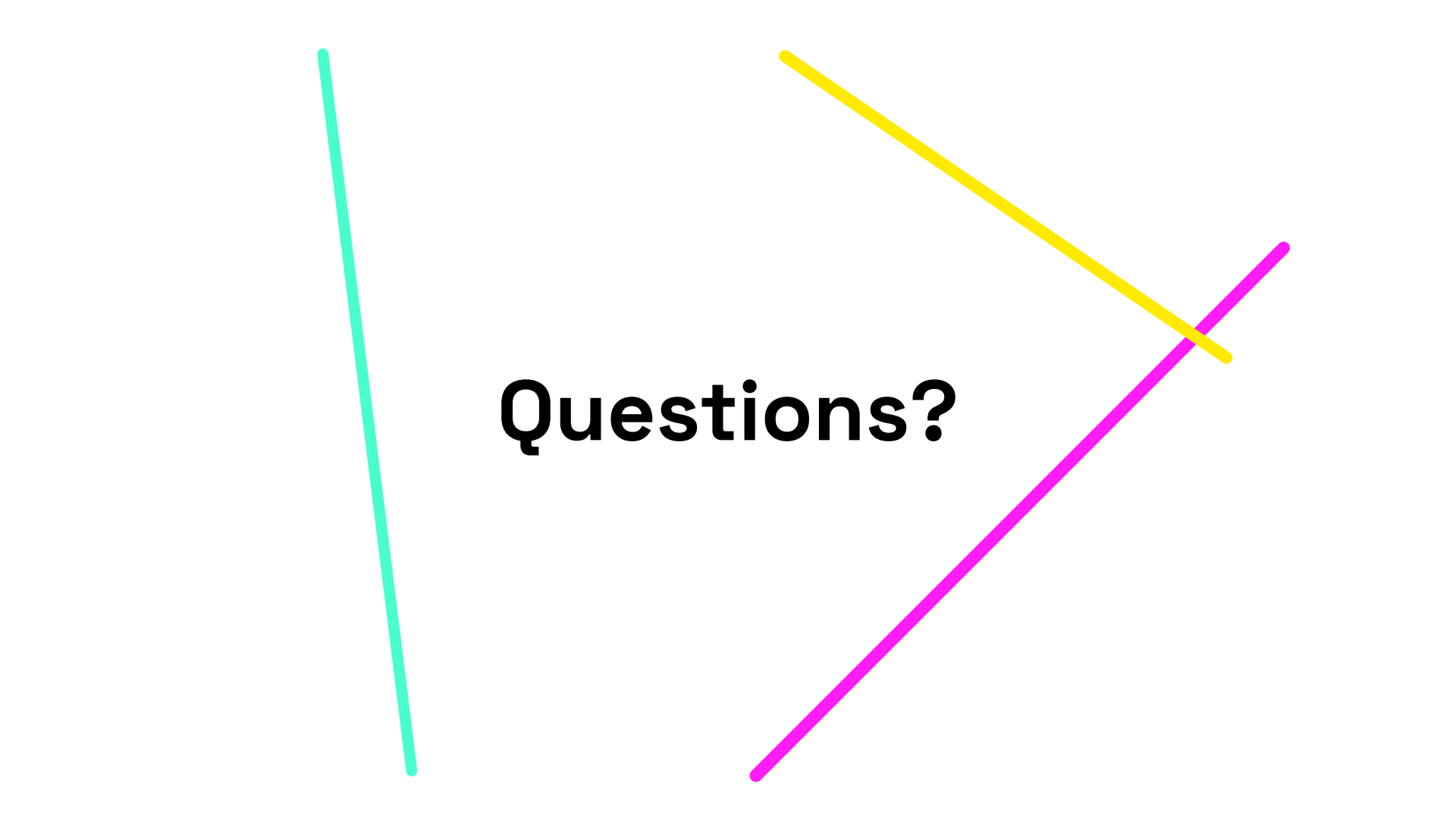
- Detect issues at compile time
- Possible false positives
- Code conversion cost
- Disabled by turning off warning at call site
- Dependent on clang 20+

Both tools were designed to complement each other

Use both, and write better real-time code!

Thanks!

- Doug Wyatt
- Ali Barker
- Reviews on LLVM work
  - @vitalybuka
  - @maskray
  - @sirride
  - @AaronBallman
  - ... many more ...
- Slide review and contributions
  - Ryan Avery
  - Oliver Graff
  - Stuart Glaser


Questions?

Real-time programming meetup

CppCon 2024
6pm Wednesday
Mountain Pass Sports Bar

Anyone welcome!

Help improve RealtimeSanitizer

- Adding support
  - Windows
  - Android
  - Other architectures (than x86-64 and arm64)
- Adding new interceptors
- Testing out experimental unbound loop checking on your codebase
- General feedback/testing

We are happy to help you get started!
Find us on Discord

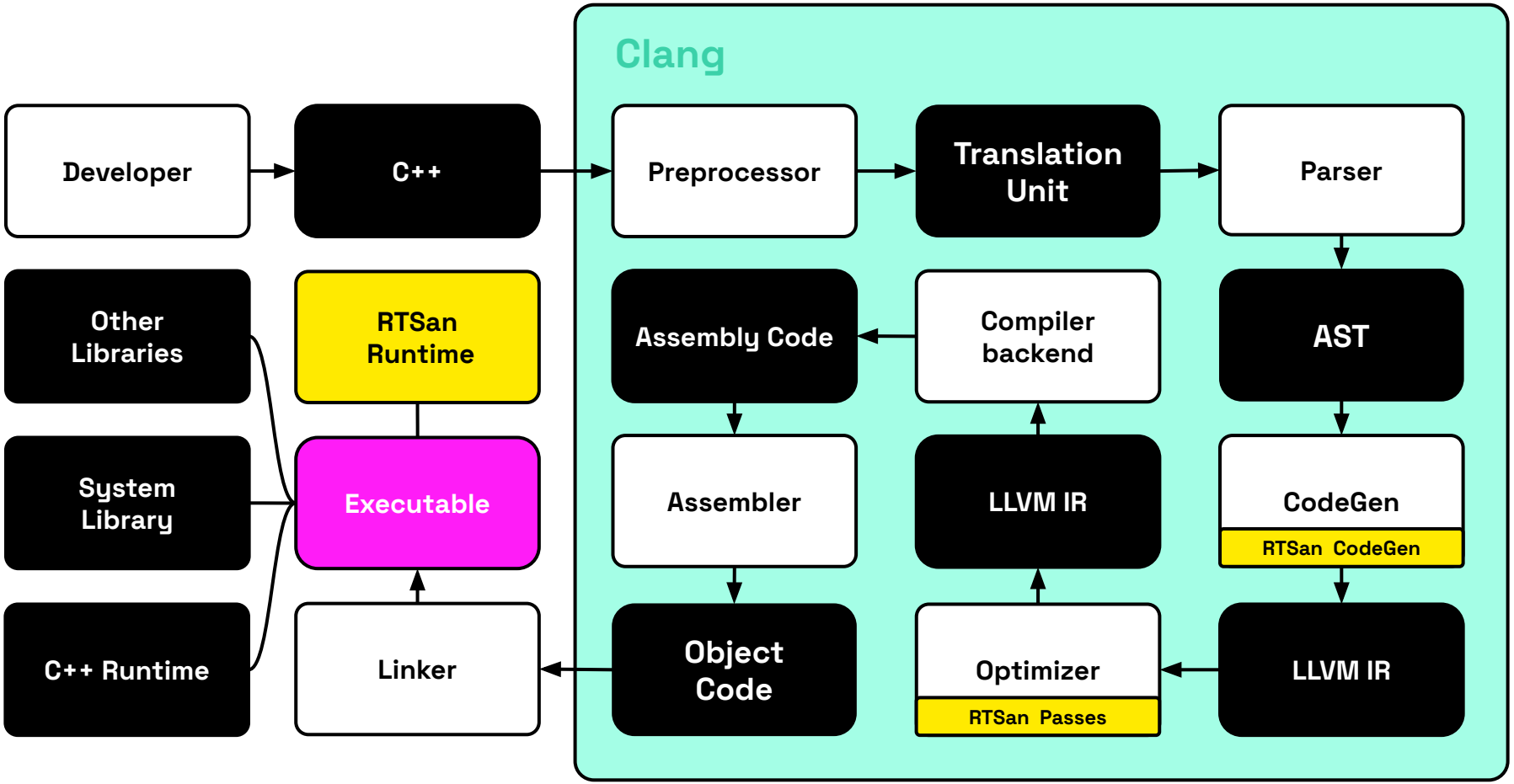
Appendix

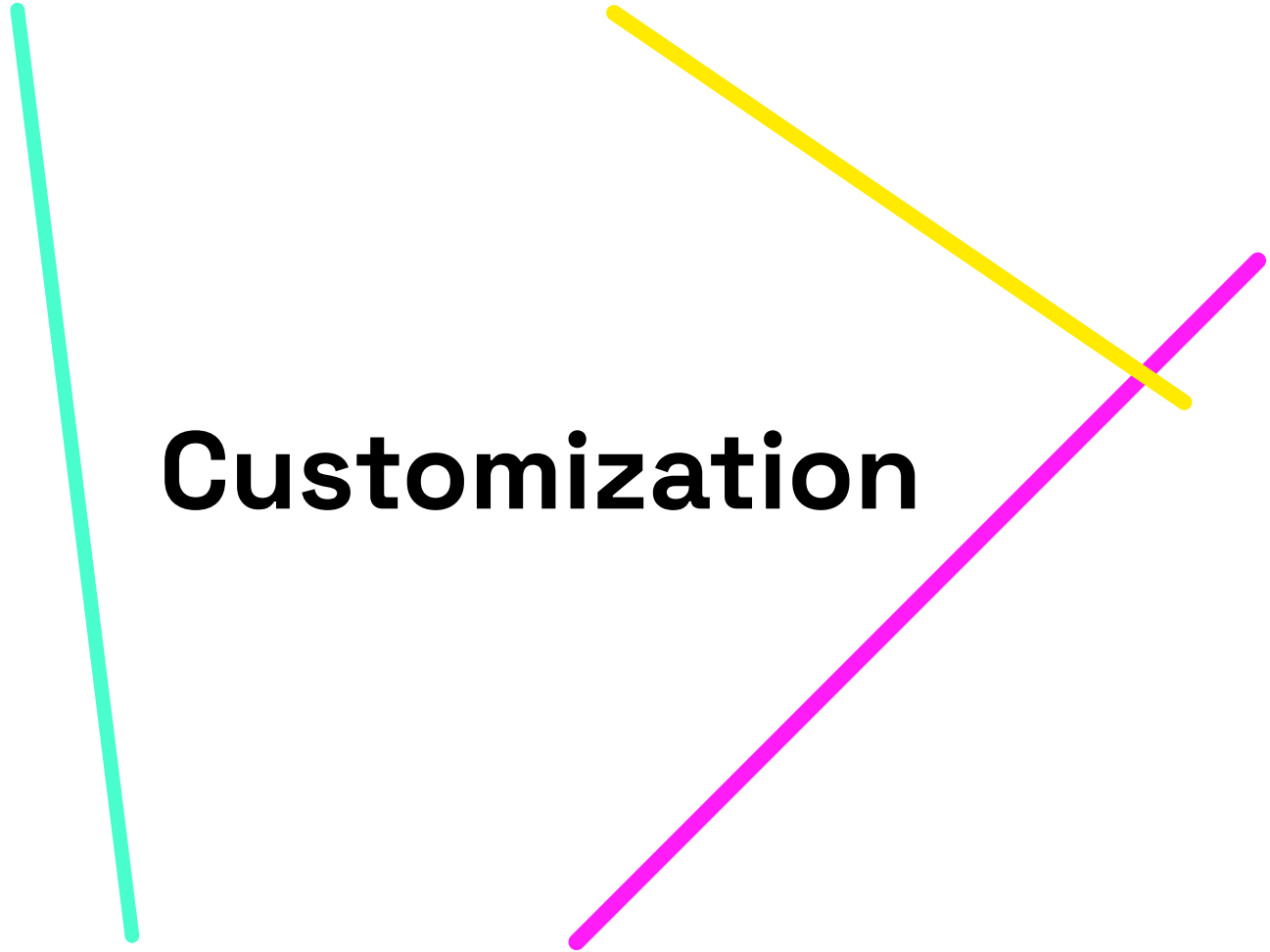
What about DTrace?

- DTrace does a lot of similar things to RTSan
- However we wanted to push for RTSan in a central repository, where there was “one source of truth” for real-time safety
  - Duplication of “intercepted functions” across companies
  - Different norms marking what is real-time context
    - Can do this with function names, or thread priority
- DTrace also requires bypassing security on OSX, which is sometimes disallowed in some IT departments
- We want every platform to be supported trivially
- Sanitizer approach offers extra benefits like scoped disabling

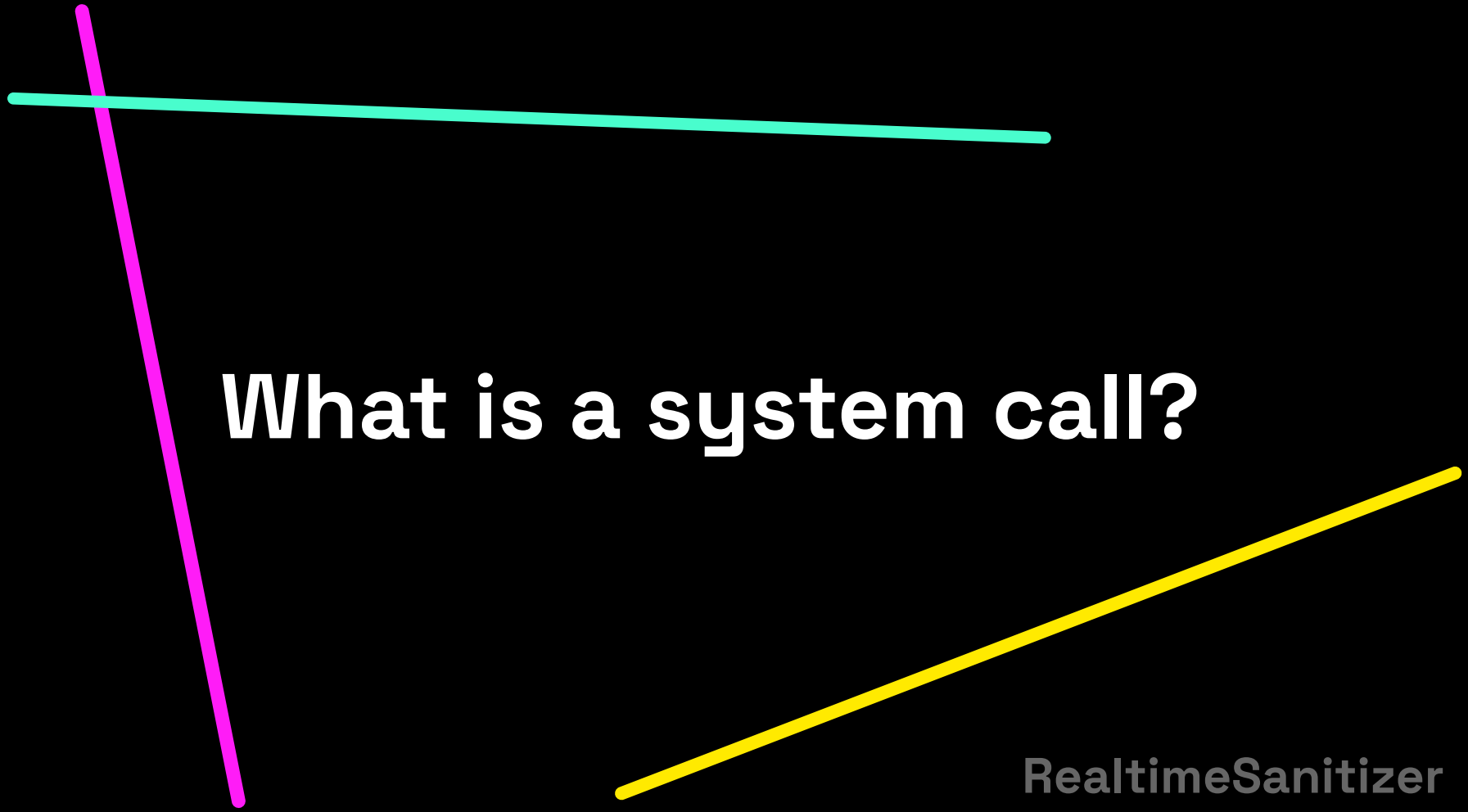
RealtimeSanitizer road map

- Soon
  - More interceptors
  - User defined [[blocking]] functions
  - Better interop with existing sanitizers (RTSAN_OPTIONS)
  - Adding “continue mode”
  - Suppression lists
- Later
  - Detection of nondeterministic loops
- Someday?
  - Windows support
  - Android/iOS support
  - Other architectures beyond Arm64 and x86_64

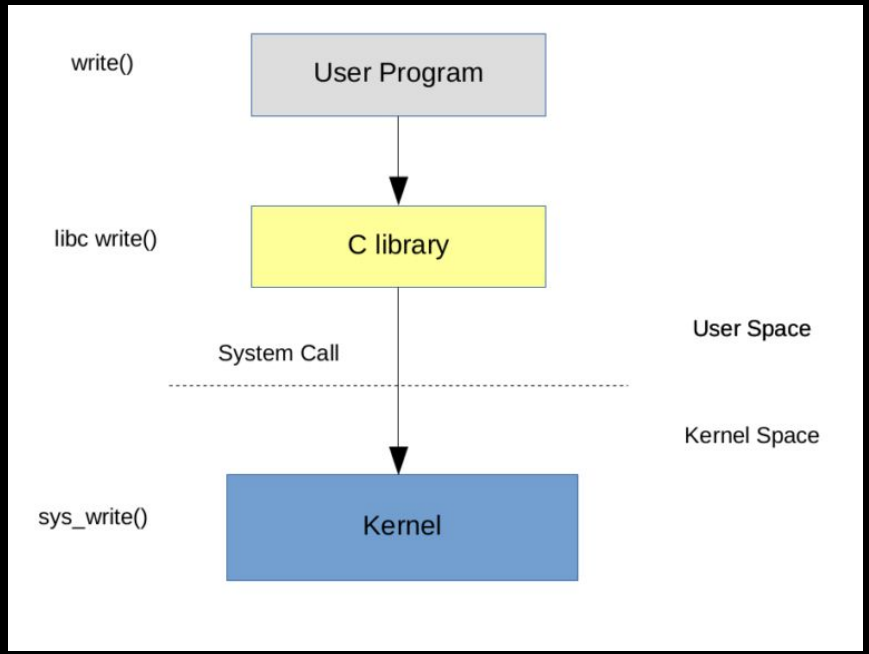

Clang
Developer
C++
Preprocessor
Translation Unit
Parser
Other Libraries
RTSan Runtime
Assembly Code
Compiler backend
AST
System Library
Executable
Assembler
LLVM IR
CodeGen
RTSan CodeGen
C++ Runtime
Linker
Object Code
Optimizer
RTSan Passes
LLVM IR


Customization

“system call” ?


What is a system call?
RealtimeSanitizer

A system call...



- Is a request to the operating system for interaction with a shared resource
- Has a specific number associated with them, which is passed to the processor via a register.
- Is not (typically) made directly from user code, but made from a wrapper library like libc.

macOS

Interception is directly supported on macOS, using a feature called **dyld interposing**.

When loading any Mach-O binary, **dyld** treats:

- a pair of function pointers in a `__DATA,__interpose` section…
- …as an instruction to **re-route all calls** from one function to the other

Calling the original function is simply a case of calling the original!

```
struct interpose_substitution
{
    const size_t ptr_to_replacement_function;
    const size_t ptr_to_original_function;
};

INTERCEPTOR (void *, malloc, size_t size)
{
    log_system_call (“malloc”);
    return REAL (malloc) (size);
}

// expands to…

extern "C" void * malloc (size_t size);
extern "C" void * wrap_malloc (size_t size);

const interpose_substitution substitution_malloc[]
    __attribute((section("__DATA, __interpose"))) = {
    {
        reinterpret_cast<const size_t> (wrap_malloc),
        reinterpret_cast<const size_t> (malloc),
    },
};

extern "C" void * wrap_malloc (size_t size)
{
    log_system_call (“malloc”);
    return malloc (size);
}
```

Linux

It's more straightforward to replace a function on Linux. If the function symbol is defined in your executable, that'll get found "first".

But calling the original (replaced) function is a little trickier!

It's possible to use **`dlsym`** to find the "next" occurrence of the function symbol in the search order.

```
INTERCEPTOR (void *, malloc, size_t size)
{
    log_system_call ("malloc");
    return REAL (malloc) (size);
}

// expands to…

typedef void * (*malloc_function_type) (size_t size);

namespace __interception {
malloc_function_type real_malloc;
}

void * malloc (size_t size)
{
    log_system_call ("malloc");
    return __interception::real_malloc (size);
}
```

```
INITIALISE_INTERCEPTOR (malloc);

// expands to…

__interception::real_malloc = dlsym (RTLD_NEXT, "malloc");
```

Must be assigned before first call to malloc!

C++ → LLVM IR → Binary

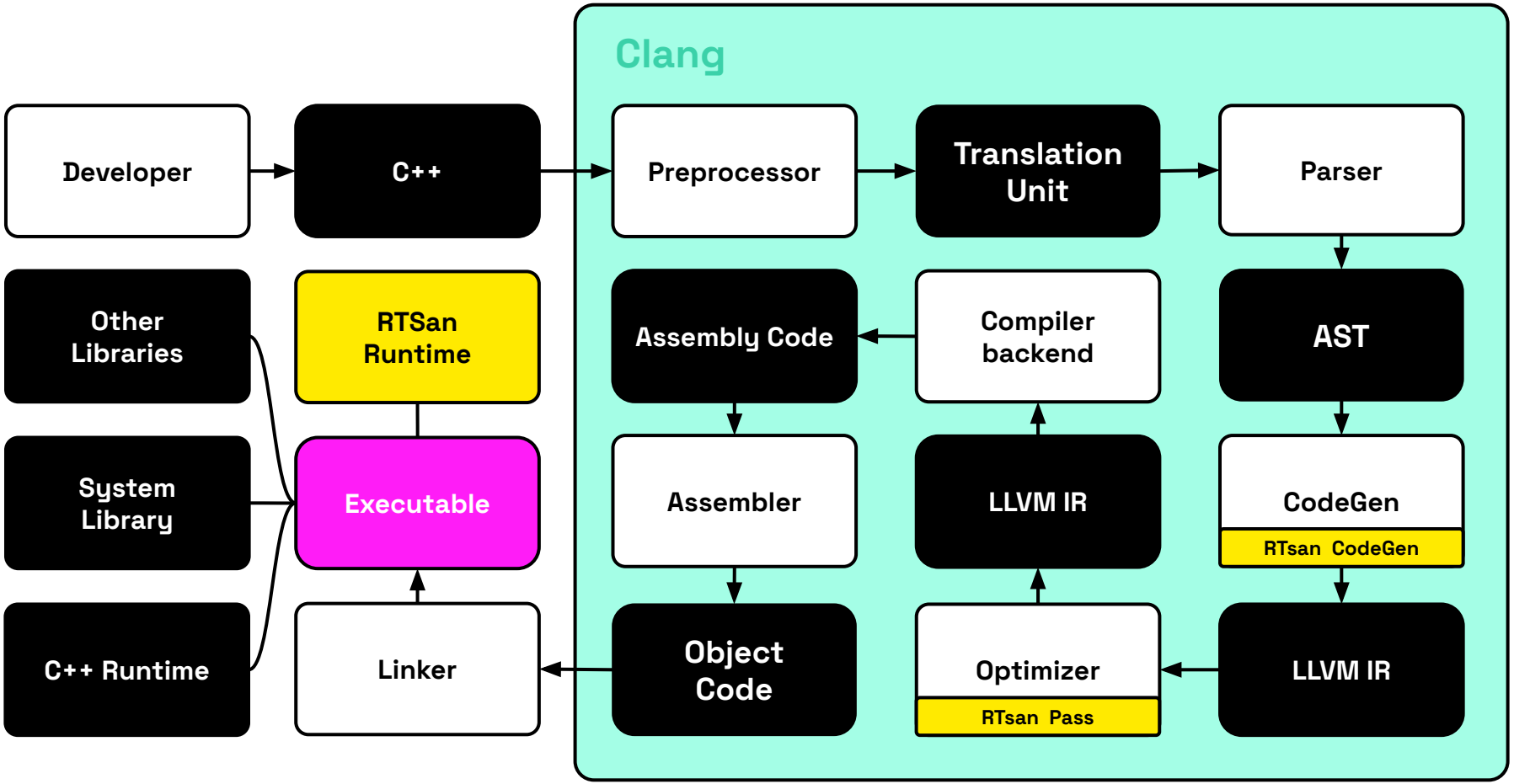
```
auto exit_code()
{
    return 0;
}

int main()
{
    return exit_code();
}
```

```
define i32 @_Z9exit_codev() #0 {
  ret i32 0
}

define i32 @main() #1 {
  %1 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  %2 = call i32 @_Z9exit_codev()
  ret i32 %2
}
```

```
cffa edfe 0c00 0001 0000 0000 0200 0000  ................
1000 0000 e802 0000 8500 2000 0000 0000  .......... .....
1900 0000 4800 0000 5f5f 5041 4745 5a45  ....H..__PAGEZE
524f 0000 0000 0000 0000 0000 0000 0000  RO..............
0000 0000 0100 0000 0000 0000 0000 0000  ................
0000 0000 0000 0000 0000 0000 0000 0000  ................
0000 0000 0000 0000 1900 0000 e800 0000  ................
5f5f 5445 5854 0000 0000 0000 0000 0000  __TEXT..........
0000 0000 0100 0000 0040 0000 0000 0000  .........@......
0000 0000 0000 0000 0040 0000 0000 0000  .........@......
0500 0000 0500 0000 0200 0000 0000 0000  ................
5f5f 7465 7874 0000 0000 0000 0000 0000  __text..........
5f5f 5445 5854 0000 0000 0000 0000 0000  __TEXT..........
783f 0000 0100 0000 2800 0000 0000 0000  x?......(.......
783f 0000 0200 0000 0000 0000 0000 0000  x?..............
0004 0080 0000 0000 0000 0000 0000 0000  ................
5f5f 756e 7769 6e64 5f69 6e66 6f00 0000  __unwind_info...
5f5f 5445 5854 0000 0000 0000 0000 0000  __TEXT..........
a03f 0000 0100 0000 6000 0000 0000 0000  .?......`.......
a03f 0000 0200 0000 0000 0000 0000 0000  .?..............
0000 0000 0000 0000 0000 0000 0000 0000  ................
1900 0000 4800 0000 5f5f 4c49 4e4b 4544  ....H..__LINKED
4954 0000 0000 0000 0040 0000 0100 0000  IT.......@......
0040 0000 0000 0000 0040 0000 0000 0000  .@.......@......
0802 0000 0000 0000 0100 0000 0100 0000  ................
0000 0000 0000 0000 3400 0080 1000 0000  ........4.......
0040 0000 3800 0000 3300 0080 1000 0000  .@..8...3.......
3840 0000 4800 0000 0200 0000 1800 0000  8@..H...........
8840 0000 0300 0000 b840 0000 3000 0000  .@.......@..0...
0b00 0000 5000 0000 0000 0000 0000 0000  ....P...........
0000 0000 0300 0000 0300 0000 0000 0000  ................
0000 0000 0000 0000 0000 0000 0000 0000  ................
0000 0000 0000 0000 0000 0000 0000 0000  ................
0000 0000 0000 0000 0000 0000 0000 0000  ................
0e00 0000 2000 0000 0c00 0000 2f75 7372  .... ......./usr
2f6c 6962 2f64 796c 6400 0000 0000 0000  /lib/dyld.......
1b00 0000 1800 0000 80ae 897c 33d6 3c11  ...........|3.<.
843c a99f d7b9 9f38 3200 0000 2000 0000  .<.....82... ...
0100 0000 0000 0e00 0000 0e00 0100 0000  ................
0300 0000 0007 f703 2a00 0000 1000 0000  ........*.......
0000 0000 0000 0000 2800 0080 1800 0000  ........(.......
803f 0000 0000 0000 0000 0000 0000 0000  .?..............
```


Clang
Developer
C++
Preprocessor
Translation Unit
Parser
Other Libraries
RTSan Runtime
Assembly Code
Compiler backend
AST
System Library
Executable
Assembler
LLVM IR
CodeGen
RTsan CodeGen
C++ Runtime
Linker
Object Code
Optimizer
RTsan Pass
LLVM IR

By detecting calls to libc wrapper functions, we can warn about potential syscalls