Safety at Scale: Advancing Safety with 100s MLoC of C++

AsiaLLVM 2025

Kinuko Yasuda (Google)

Jun 10, 2025

Memory Safety, and C++’s (remarkable) unsafeness

Amount of the concerns, in numbers

- **50 years** of memory unsafety, causing C++ services to crash and be attacked
- Our internal analysis estimates **75%** of CVEs used in zero-day exploits are memory safety vulnerabilities
- In 2025, we keep seeing CVEs in the industry because of memory safety issues (e.g. CVE-2025-1414, CVE-2025-22457)

Most (if not all)* of these would not have happened in the first place if they were written in memory-safe languages, e.g. Rust

*) caveat JIT cases, unsafe part of MSL etc

The Problem Space in Our Scope

Memory Safety

- **Spatial safety** — Out-of-bounds data access
- **Temporal safety** — Use-after-free, use-after-return, double free, ...
- Type safety
- Initialization safety — Use of uninitialized memory
- Data-race safety

- Nullptr safety — Dereference of nullptr
- Integer overflow safety
- more!

Many forms of UB, many forms of safety issues

Scale of C++ at Google

Lines of code	`O(1,000,000,000)`
Pointer declarations	`O(10,000,000)`
Source files	`O(1,000,000)`
Production services	`O(100,000)`
Developers	`O(10,000)`

Google’s approaches for Memory Safety

Bug detection techniques like sanitizers and fuzzing allowed us to see the size of the problem, however they are not enough to move the needle for **O(1B) LoC of C++**

Two-pronged approach:

- Progressively and consistently adopt memory-safe languages (MSLs) in new development wherever possible
- Retrofitting safety to our existing C++ codebase, with a stronger focus on **prevention** and **hardening at scale** ⬅️ **today’s focus**

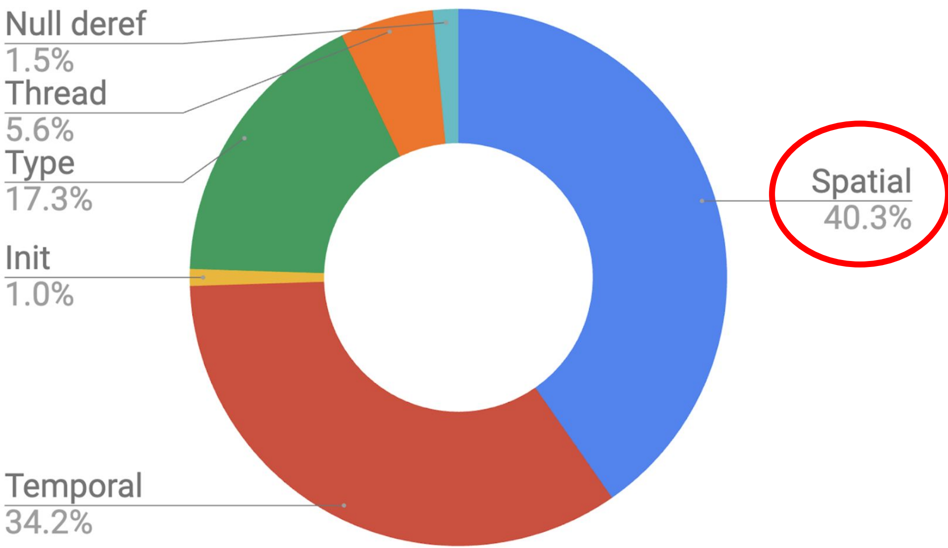
Spatial safety: Retrofitting spatial safety to our C++ code base

Temporal safety: Pushing the boundaries of limitations for C++ safety

Spatial safety issues in C++

```
int foo(const vector<int>& v) {
   ... // complex offset computation
   return v[offset];
}
```

```
void bar(int *p) {
  ...
  int x = p[i];
  ...
}
```



Classic, but pretty common, and heavily exploited

Simple solution, big challenges

If the memory access is outside the intended range, we'll get UB…

Solution: Insert runtime bounds checking

```
int foo(const vector<int>& v)
    ... // cryptic offset comput
    return v[offset];
}
```

Implement bounds checking in common data structures, e.g. in std::vector

Libc++ Hardened Mode

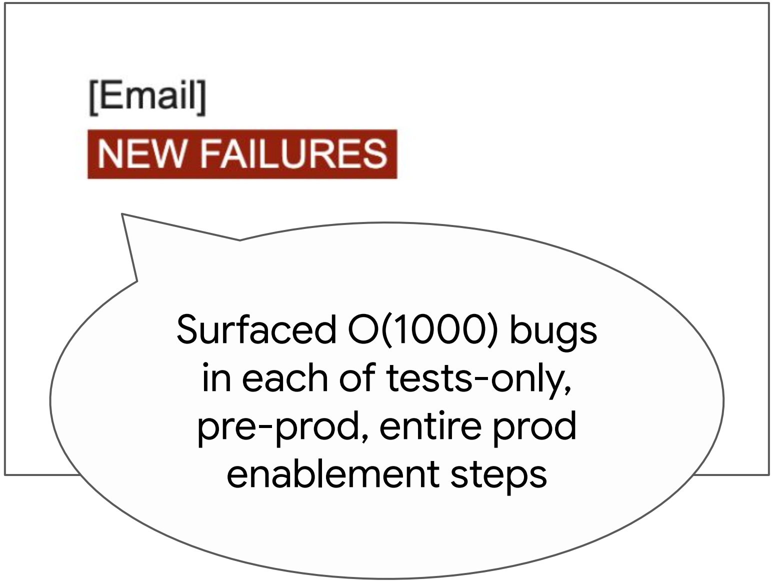
Challenge: communicate why it makes sense despite the concerns of

- potential increase of deterministic crashes (that were previously heisenbugs)
- additional overhead for bounds-checking

across **O(10^5) production services** through measurement

Small steps... towards eventually enabling it by default

First attempt made in late 2022, enabling it in development build only



1. Enabled in tests only
2. Baked in pre-prod, conducted performance evaluations
3. Piloted with a small set of prod services
4. Eventually expanded to **our entire infrastructure** (2024)
   E.g. *Search, Gmail, Drive, YouTube, Maps, etc*

Results and observations in retrospect

Our speculation: We thought that bounds checking would be too expensive for production deployment

> Just to emphasize: **I was (very) wrong.**

https://chandlerc.blog/posts/2024/11/story-time-bounds-checking/

The results: only **0.3% performance impact** across our services on average

Results and observations in retrospect

Our speculation: We thought that bounds checking would be too expensive for production deploy

Increased
awareness t
need these che
production b

MSVC had added
such checks long
time ago, Apple
drove LLVM RFC for
saf

Multiple languages
with bounds checking
drove improvements in
LLVM

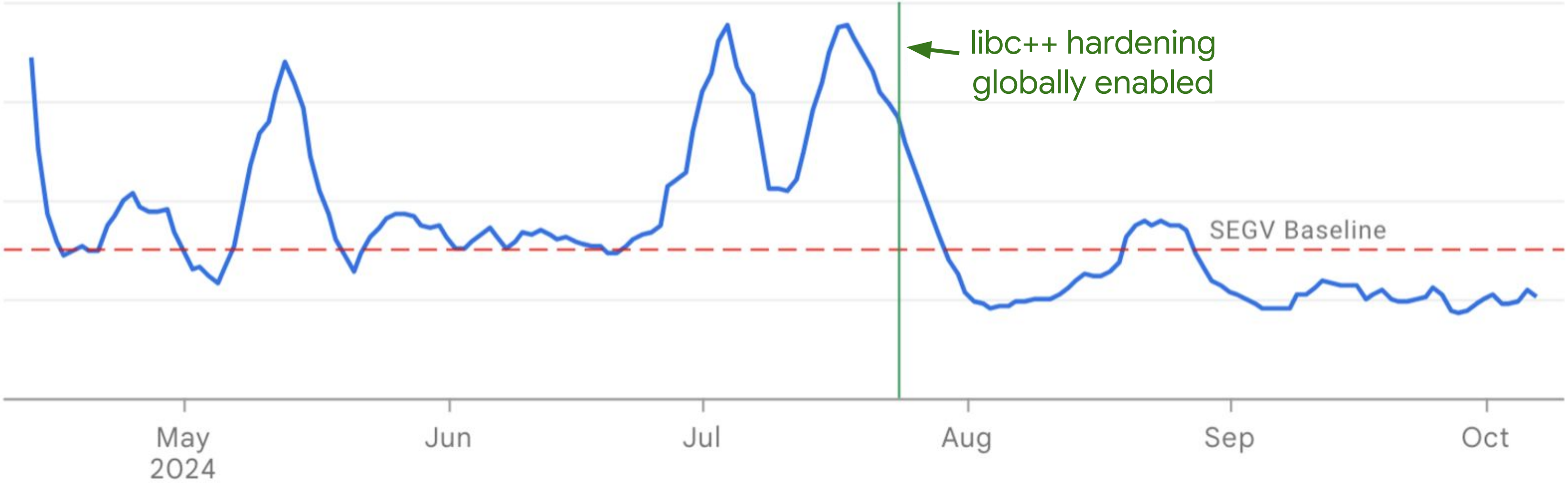
Apple's performant libc++
hardening impl work, we
also contributed a fix that
pushed 0.35% to 0.3%!

Profile-guided
optimizations (PGO)
for another big
difference

The results: only **0.3% performance impact** across our services on average

Impact: preventing exploits, improved reliability

Disrupted or **would have prevented** internal red team exercises
Uncovered over **1,000 bugs**, would prevent 1000-2000 bugs each year



30% reduction in Segmentation faults across our fleet

Next steps: Safe Buffers, and enable more hardening

- Following Apple's RFC for safe buffers in C++
- "Automated" migration of code to use bounds-checkable containers
  - Reminder: O(10^7) pointers in our codebase!
  - Challenge: Where a buffer is built/decl'd, and where it is used, are often different
  - Needs to build pointer-flow graphs for each TU, and combine them to solve across codebase
- Combined with compiler-assisted bounds checking where possible

Expect to be in a much better position for spatial safety in coming years!

Spatial safety: Retrofitting spatial safety to our C++ code base

Temporal safety: Pushing the boundaries of limitations for C++ safety

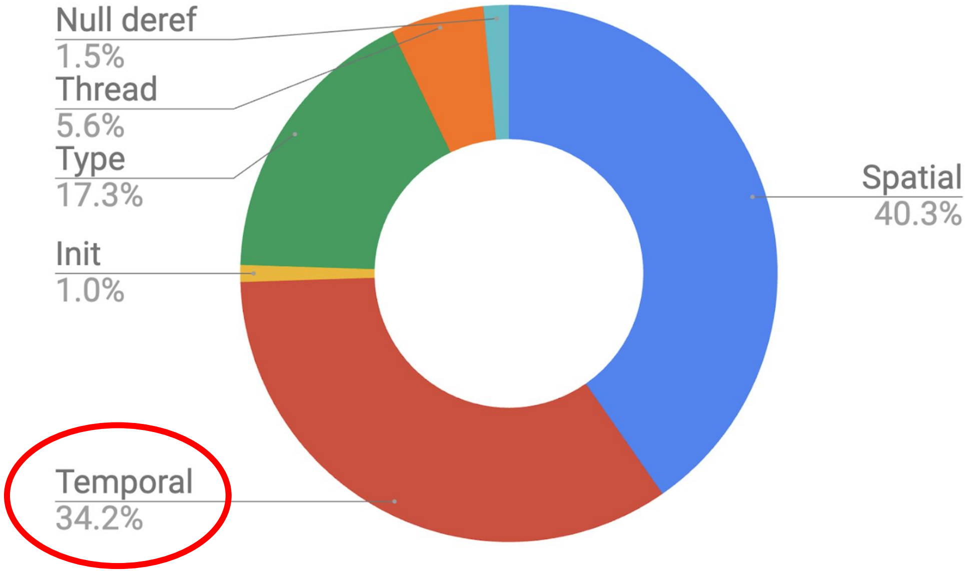
Temporal safety issues in C++

```
Object* obj_;
obj_ = new Object();
```

```
void freeObj() {
  delete obj_;
}
```

```
void useObj() {
  obj_->doSomething();   // ⚠️UaF
}
```

```
std::string_view getSV() {
  std::string local = "stack";
  return local;  // ⚠️UaR
}
```



Yet another classic bug category, but huge source of crashes and exploits

Various solutions proposed… many involve runtime cost

Runtime mechanism and/or checks

- Garbage-collection
- Ref-counting
- Lock and key
- Zapping and quarantining, possibly combined with other techniques
- Never-free memory
- …

The cost has become significantly cheaper, but still n times more expensive than bounds checking

Static solution: borrow checking?

- Semantics of “Ownership” concept in safer dialects of C/C++ eventually evolved as “Borrow checking” in Rust
- Temporal safety errors can be **statically avoided** at compile-time

single **exclusive** **mutable** reference

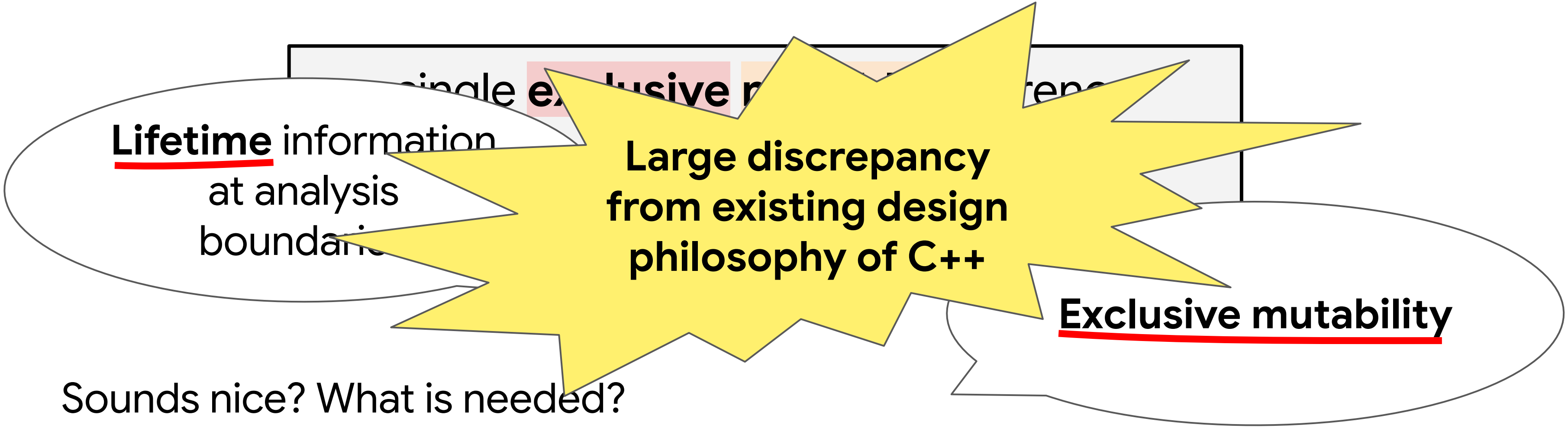
$\oplus$

multiple **shared** **immutable** references

Sounds nice? What is needed?

Static solution: borrow checking?

- Semantics of "Ownership" concept in safer dialects of C/C++ eventually evolved as "Borrow checking" in Rust
- Temporal safety errors can be **statically avoided** at compile-time



Sounds nice? What is needed?

Let’s step back... and see what might still be possible

Long story short: we explored the solution space in Clang extensions, with a focus on **deployability** and **safety benefits** in existing C++ code

We started with: `lifetimebound` and `gsl::Pointer`

- Community support: available since Clang 7 (2018~), MSVC 17.7 (2022~)
- Limited expressivity, however can tell lifetime contracts for subset cases of what Rust’s lifetimes or [RFC] Lifetime annotations for C++ can cover
- **These can add information that is lacking in C++ code otherwise**

What the analysis could cover, and what it didn't

Can catch simple, statement-local violations in initialization cases like:

```
std::string_view sv = absl::StrCat("Hello!", "world");
```

```
std::span<const int> ints = { 17, 19 };
```

Looks promising! However it couldn't detect simpler variations like following:

```
std::string_view sv;
sv = absl::StrCat("returns", "temporary");
```

assignment case 😣

```
const Object* obj_;
obj_ = std::make_unique<Object>(var).get();
```

assignment case 😣

```
std::optional<string_view> osv = tmpString();
```

Nested case 😣

Extended the coverage of analysis in Clang 17+

Can catch simple, statement-local violations in initialization cases like:

```
std::string_view sv = absl::StrCat("Hello!", "world");
```

```
std::span<const int> ints = { 17, 19 };
```

We improved the analysis implementation, and they can be all caught today:

```
std::string_view sv;
sv = absl::StrCat("returns", "temporary");
```

assignment case ☺️

```
const Object* obj_;
obj_ = std::make_unique<Object>(var).get();
```

assignment case ☺️

```
std::optional<string_view> osv = tmpString();
```

Nested case ☺️

More cases that are newly supported in Clang 20+

returning stack address

```
std::span<int> makeSpan() {
   int local[3] = {1, 2, 3};
   return std::span<int>(local);
}
```

standard container cases

```
void test() {
  std::vector<std::string_view> vsv;
  for (int i = 0; i < kBatchSize; ++i)
    vsv.push_back(absl::StrCat("/batch/", i));
}
```

Initialization from gsl::Owner

```
struct S2 {  std::string s2; };
void test() {
  std::string_view v = S2().s2; // dangling
}
```

Limitation: can only express limited relationships

`lifetimebound` can only cover the following contracts:

- A parameter of a function is referenced by its return value
- `this` object is referenced by the return value of its member function

A "is referenced by" B ⇒
A "should outlive" B

But can't express the following cases:

- A parameter of a member function is referenced by `this` object
- A parameter of a function is referenced by another parameter or a global variable

x is captured by 'this'

```
struct S {
  void capture(const std::string& x) { this->x = x; };
  std::string_view x;
};

void test(S& s) {
   s.capture(createTmpString());  // ⚠️'s' captures a reference to a temporary
}
```

lifetime_capture_by for "referenced (captured) by X"

To express "a parameter is referenced (=~ captured) by X" cases

```
struct S {
    void set(const std::string& x [[clang::lifetime_capture_by(this)]]) {
        this->x = x;
    };
    std::string_view x;
};
```

captured by 'this'

```
void addToSet(std::string_view s [[clang::lifetime_capture_by(set)]],
              std::set<std::string_view>& set) {
    set.insert(s);
}
```

captured by another parameter

```
std::set<std::string_view> globalSet;
void addToGlobalSet(std::string_view s [[clang::lifetime_capture_by(global)]]) {
    globalSet.insert(s);
}
```

captured by a global variable

Deployment

Steps we followed:

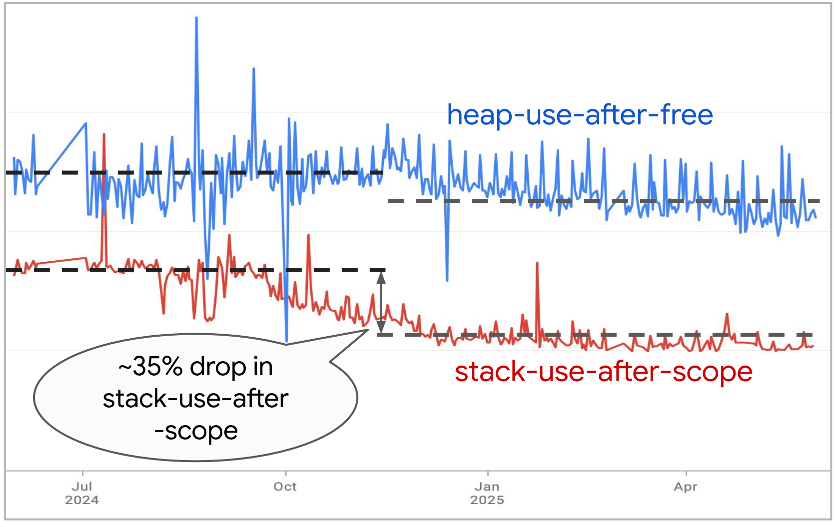
- Added the annotations to a set of key core libraries (**absl** and others)
- Fixed O(1000) existing violations across our codebase
- Enabled all of the improved analyses **as warning-as-error by default**

Similar amount of breakages, but **much easier** than libc++ hardening cases:

- All breakages happen at compile-time, not after they hit production
- Very obvious error locations and reasons; no need to decrypt stack traces
- No performance concerns

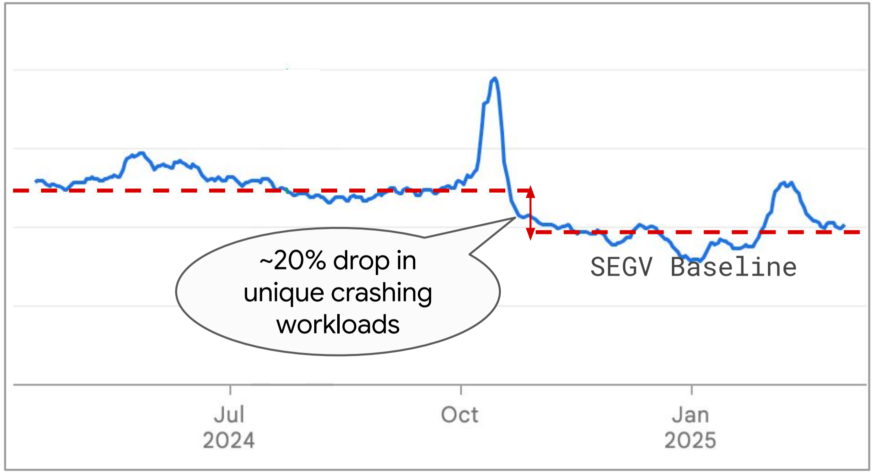
One of other learnings: there could be complicated UaFs, but many were also simple ones-- in retrospect it feels funny how easy it was to introduce such errors

Impact: preventing violations, improved reliability



- Reduced **~35%** of use-after-scope in tests
- **~17%** of UaFs in production (GWP-Asan reports) would have been prevented
- Surfacing **O(100) warnings per week** in the IDE, preventing future UaFs from landing

Another push for the improved reliability: **~20% drop** of unique crashing workloads (moving average) with Segmentation faults



Next steps: flow-sensitive, intra-procedural analysis

Current analysis doesn't track lifetimes beyond single statement:

```
std::string_view foo() {
  std::string local = "local";
  std::string_view view = "default";
  view = local; // No warning!
  return view;  // ⚠️
}
```

```
std::string_view bar(bool cond) {
  std::string_view view = "default";
  if (cond) {
    std::string local = "local";
    view = local;     // No warning!
  }
  return view;        // ⚠️
}
```

- Proposing **a new analysis implementation** that performs CFG-based, flow-sensitive points-to analysis inspired by the Rust's borrow checker (Polonius) https://discourse.llvm.org/t/rfc-intra-procedural-lifetime-analysis-in-clang/86291
- Still sticking to existing annotations (i.e. lifetimebound), however wishing to lay a better foundation that can also support Rust-like lifetime annotations if/when Clang adopts it

Wrap up

Advancing safety with the unsafe reality of **100s MLoC of C++**

- Invest in the prevention and hardening, where the benefits accrues as **future violations can also be prevented**
- Adopting and retrofitting the learnings from Memory-safe languages
  - **For Spatial safety:** Default-enabled bounds checking everywhere
  - **For temporal safety:** Lifetime information & analysis
  - For initialization safety: Initialize ALL the things
  - For nullptr safety: Pointers should be able to state nullability contract
- **Outcome**
  - Surfaced and fixed O(1000) existing violations
  - Would keep preventing O(1000) violations each year
  - Had observable reliability wins
  - Demonstrated security effectiveness (against internal offense exercise)

Resources & RFCs: find us more at

Technical resources and RFCs

- RFC: C++ Buffer Hardening https://discourse.llvm.org/t/rfc-c-buffer-hardening/65734
- RFC: Lifetime annotations for C++ https://discourse.llvm.org/t/rfc-lifetime-annotations-for-c/61377
- Lifetime analysis improvements https://discourse.llvm.org/t/lifetime-analysis-improvements-in-clang/81374
- RFC: Introduce [[lifetime_capture_by(X)]] https://discourse.llvm.org/t/rfc-introduce-clang-lifetime-capture-by-x/81371
- RFC: Intra-procedural analysis in Clang https://discourse.llvm.org/t/rfc-intra-procedural-lifetime-analysis-in-clang/86291

If you're interested in discussing these topics more with us, join the discussions at https://discourse.llvm.org/t/rfc-forming-llvm-working-group-on-memory-safety/84434

Thank you ☺️

Credits to all the active contributors who have been driving this area:

Utkarsh Saxena
Haojian Wu
Yitzhak Mandelbaum
Max Shavrick
Alex Rebert
Christopher Di Bella
Dmytro Hrybenko
Martin Brænne
Ilya Biryukov
Chandler Carruth
Gábor Horváth

... and many more!