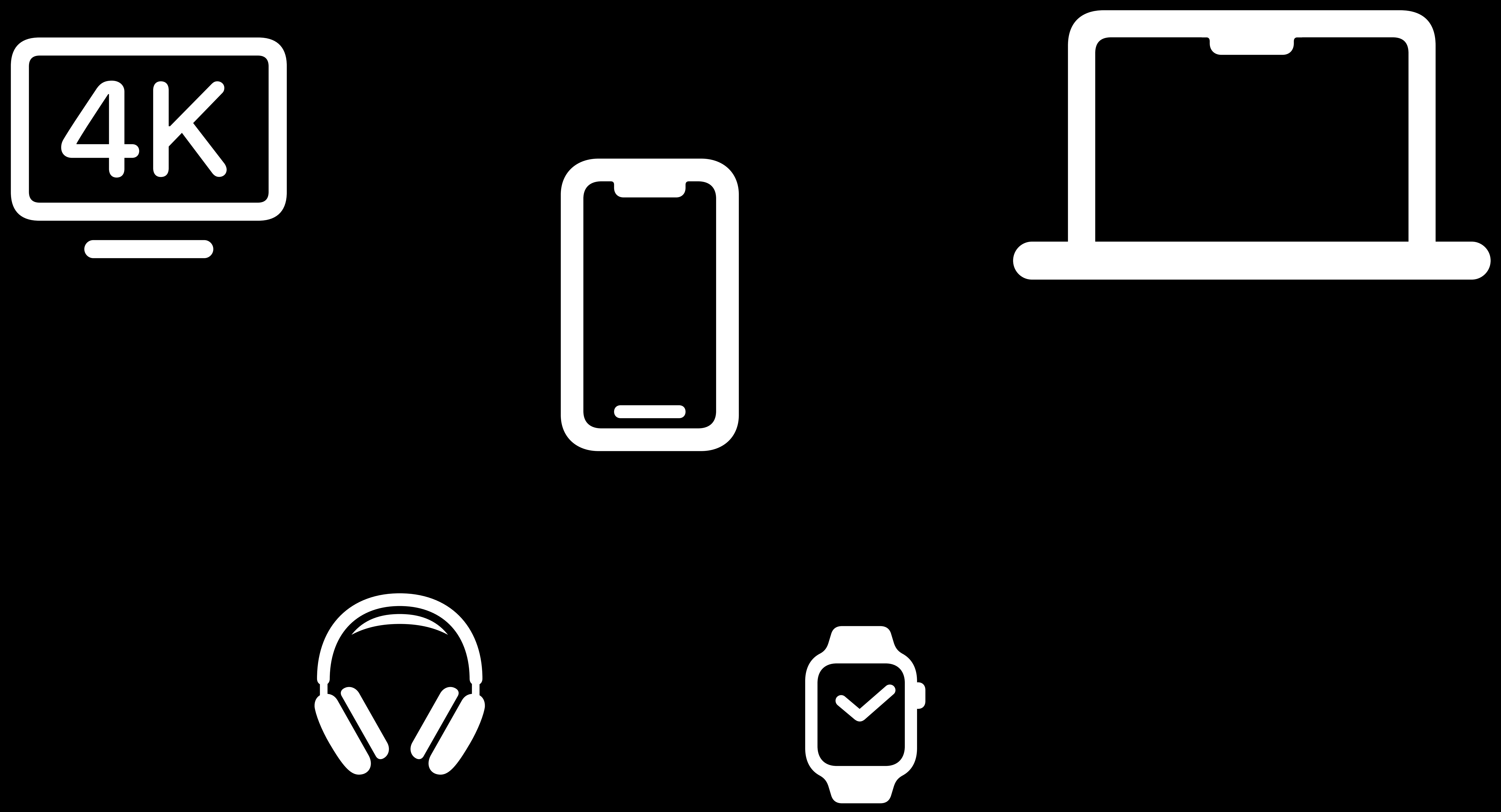
Eliminating Entire Classes of Memory Safety Vulnerabilities in C and C++

Devin Coughlin

EuroLLVM 2025


4K

Security attacks are on the rise, threatening both financial and physical safety

Memory safety is a critical challenge

Memory Safety Strategy

- Full **memory safety requires** using **a memory-safe language**
- There is **too much code to rewrite it <u>all</u>**
- Adopt memory safe languages in **new code**
  - **Rewrite high-value** codebases in a memory safe language

Metaphor: Islands of Memory Safety

- **Islands** of memory safe code…
- In a **vast ocean** of memory-unsafe C and C++.

Must protect ocean of **large existing C/C++ codebases**

Cannot Make C/C++ Fully Memory Safe

- But can aim to **eliminate entire classes of vulnerabilities**
- If we can’t eliminate, **mitigate strongly**!
- **Static and dynamic bug-finding** are useful but **not comprehensive**

Definition of Success

- **Shifting attacker behavior**
  - Drastically **increase cost** of attack development
  - **Attackers move on** to other vulnerability classes or unprotected codebases

- Ultimate sign of success
  - **Memory safety** bugs are **no longer the low-hanging fruit**
  - Attackers move on to **logic bugs**

Dimensions of Memory Safety

- Guaranteed initialization
- Bounds safety
- Lifetime safety
- Type (cast) safety
- Thread safety

Guaranteed Initialization

Guaranteed Initialization for C and C++

- Developed `-ftrivial-auto-var-init` extension in Clang to **protect stack**
  - Guarantee **initialization to zero**
  - Prevents **information disclosure** and many **stack grooming** attacks
- **Zero-initialize on** `free()` to protect **heap**
- Deployed on **hundreds of millions of lines of code**
- Not perfect: **Zero not intended** initial value in **~20%** of cases
- **Still pretty good!**

More Information

LLVM RFC

https://discourse.llvm.org/t/rfc-automatic-variable-initialization/50327

Bounds Safety

Bounds Safety for C

- Developed `-fbounds-safety` extension for C
- Users specify buffer/bounds relations with **code annotations**
- **Run-time bounds checks** trap on out-of-bounds memory accesses
- **Compile-time rejection** of code if **bounds not determinable at run time**
- Made key design choices to lower adoption cost
  - Time to adopt: ~1 hour per 2,000 LOC
  - Adopted in **millions lines of C code**

Parameter annotations for incremental adoption

Code modification to note that `count` is the element count of `buf`

C

```
void fill_array_with_indices(int *buf __counted_by(count), size_t count) {
  for (size_t i = 0; i <= count; ++i) {
    if (i < 0 || i >= count) trap();
    buf[i] = i;
  }
}
```

Compiler-generated bounds check
based on the information provided by the annotation

Use of annotations allows **preserving binary interface and signature**

Implicit wide pointers reduce annotation burden

```
void foo(int *a __counted_by(count),
         size_t count) {
  int *local = a;
  ...
}
```

Compiler Transform

```
// Internal representation
typedef struct {
  int *ptr; // Pointer value
  int *ub;  // Upper bound
  int *lb;  // Lower bound
} wide_ptr;

void foo() {
  wide_ptr local =
   {.ptr = a, .ub = a+count, .lb = a};
  ...
}
```

Local variables **implicitly carry bounds**, requiring **fewer code changes**

Light-weight annotations on ABI surface

\+

Implicit wide pointer types everywhere else

=

Incrementally adoptable bounds safety
with **reasonable annotation burden**

More Information

Clang RFC

https://discourse.llvm.org/t/rfc-enforcing-bounds-safety-in-c-fbounds-safety/70854

EuroLLVM 2023 Keynote

https://www.youtube.com/watch?v=RK9bfrsMdAM

EuroLLVM 2025 Tutorial

“Adopting -fbounds-safety in Practice”

Full Implementation

https://discourse.llvm.org/t/the-preview-of-fbounds-safety-is-now-accessible-to-the-community

(Also in process of being upstreamed to llvm.org)

Bounds Safe Programming Model for C++

- Built `-Wunsafe-buffer-usage` for C++ and **hardened libc++**
- "C++ Safe Buffers" programming model
  - Compiler **rejects raw pointer arithmetic**
  - C++ standard library provides **bounds-checked buffer abstractions** (`std::span`, `std::vector`)
- Adopted on **tens of millions of lines of code**
  - **"C written in C++"**: ~2x higher adoption cost than -fbounds-safety
  - **Modern C++**: "Zoomin'" at ~30,000+ lines of code per hour

Programmer **changes type** of `buf` to indicate it is a **contiguous span** of ints

C++

```
void fill_array_with_indices(span<int> buf, size_t count) {
  for (size_t i = 0; i <= count; ++i) {
    buf[i] = i; error: unsafe buffer access [-Werror,-Wunsafe-buffer-usage]
  }
}
```

Hardened libc++ span implementation checks bounds before indexing via **operator overloading**

More Information

Clang RFC

https://discourse.llvm.org/t/rfc-c-buffer-hardening/65734

LLVM 2023 Talks

https://www.youtube.com/watch?v=pQfjn7E4Qfc (Hardened libc++)

https://www.youtube.com/watch?v=nPRY8-FtzZg (-Wunsafe-buffer-usage)

Full Implementation in llvm.org Open Source

Lifetime Safety

Lifetime Safety: Ref-Counting Smart Pointer Analysis

- Developed Clang Static Analyzer WebKit checkers
- Enforces a **strict programming model** (not bug finding)
- **Prevents lifetime issues** caused by misuse of **reference-counted pointers**
- Adopted on **millions of lines of code**

Programmer changes type of `resource` to **indicate its lifetime** must last for entire scope

C++

```
error: Local variable 'resource' is uncounted and unsafe
Resource *resource = container->resource.get();
RefPtr<Resource> resource = container->resource.get();

doSomethingThatMightReleaseResource();

resource->use();
```

WebKit RefPtr run-time implementation guarantees lifetime until destructor is called

More Information

Full implementation in llvm.org

https://github.com/llvm/llvm-project/tree/main/clang/lib/StaticAnalyzer/Checkers/WebKit

Lifetime Safety: Type-Isolating Allocators

- Not all code uses a reference-counting discipline
- Developed **language extensions for typed allocation**
  - C: `-ftyped-memory-operations`
  - C++: `-ftyped-cxx-new-delete`
- **Mitigates use-after-free** vulnerabilities by **limiting data/pointer type confusion**
- Deployed on **hundreds of millions of lines** of userspace code
- Similar approach deployed in xnu kernel

C: Compiler Transforms Calls to Allocators

- Driven by new “typed memory operation” attribute on allocators

System Header

```
// Standard system malloc entrypoint
void *malloc(size_t size) __attribute__((typed_memory_operation(1, malloc_typed)));

// New typed malloc entry point with extra parameter for type information
void *malloc_typed(size_t size, tmo_type_descriptor_ref type_info);
```

Client Code

```
struct SomeType *allocation = malloc(sizeof(struct SomeType));
```

Compiles Into

```
struct SomeType *allocation = malloc_typed(sizeof(struct SomeType), SomeType info);
```

C++: Language support for typed allocation

- Proposal **P2719**: Type-aware allocation and deallocation functions
- Passes **type as a template parameter to operator**

C++

```
template <class T>
  requires use_special_allocation_scheme<T>
void* operator new(std::type_identity<T>, std::size_t n) { ... }

template <class T>
  requires use_special_allocation_scheme<T>
void operator delete(std::type_identity<T>, void* ptr) { ... }
```

More Information

Clang RFC

https://discourse.llvm.org/t/rfc-typed-allocator-support/79720 (For C)

https://discourse.llvm.org/t/rfc-experimental-implementation-of-p2719-type-aware-allocation-and-deallocation-functions/83876 (For C++)

LLVM 2024 Talk

https://www.youtube.com/watch?v=GGGaiGpm5BY

C++ WG21 Proposal

https://github.com/ldionne/wg21/blob/main/p2719.md

In Process of Being Upstreamed

https://github.com/llvm/llvm-project/pull/113510 (C++)

Type (Cast) Safety

C++: Static Analysis-Enforced Programming Model

- Developed WebKit MemoryUnsafeCastChecker for C++
- **Rejects unchecked casts** unless compiler can prove safe
- Works great for C++ codebases with
  - **Run-time** representation of **types**
  - Idiom of always using **run-time-checked casts**

- **Do not have a viable approach for C codebases nor all C++ codebases**

More Information

Full implementation in llvm.org

https://github.com/llvm/llvm-project/blob/main/clang/lib/StaticAnalyzer/Checkers/WebKit/MemoryUnsafeCastChecker.cpp

Thread Safety

No Strong Ideas for Thread Safety for C/C++

- **Needs more investigation**
- `-Wthread-safety` useful but does not eliminate entire class

- General recommendation for thread safety is to use thread-safe language
  - **Swift Concurrency** actor model **provides data-race freedom**

Memory Safety in Heterogeneous Codebases

Metaphor: Islands of Memory Safety

- Islands of code written in memory safe languages…
- In a vast ocean of (partially) memory-unsafe C and C++.
- **Need to protect the beaches!**

Mix of Existing Memory Safety Technologies

Swift

Fully memory safe **by design**

-fbounds-safety

Bounds-safe C language extension with **annotations**

Hardened
C++

Bounds-safe via **hardened libc++** and **-Wunsafe-buffer-usage**

Need to Preserve Safety Across Boundaries

Import `-fbounds-safety` types as
bounds-safe Swift Span

Swift

-fbounds-safety

Invoke hardened libc++ checks with Swift/C++ interop
Improve safety and ergonomics with **C++ lifetime annotations**

Guarantee bounds-safe interoperation
via strict compiler errors

See EuroLLVM 2025 talk
“C++ interoperability with
memory-safe languages”

Hardened
C++

Recap: Dimensions of Memory Safety

- **Guaranteed initialization:** `-ftrivial-auto-var-init`
- **Bounds safety:** `-fbounds-safety`, `-Wunsafe-buffer-usage` + hardened libc++
- **Lifetime safety:** typed allocators, WebKit shared pointer checks
- **Type (cast) safety**: WebKit cast static analysis, needs more investigation
- **Thread safety:** Needs more investigation

Synthesis: “Recipe” is to Enforce Strict Programming Models

Ingredients for a Programming Model

1. Compiler checkable local rules
2. Developer-provided annotations
3. Run-time support

Ingredient 1: Compiler Checkable Local Rules

- Rules **restrict** programmer to **subset of language**
  - Programming model can make **strong guarantees**
  - More **like programming language** feature than bug-finding

- Enable programmer to **reason about rules locally**
  - Programmer should be able to easily tell whether code follows or not

- Provide **tools** to **check** rules **automatically**
  - Ensure mistakes do not fall through the cracks

Ingredient 2: Developer-Provided Annotations

- Developers provide **annotations** expressing key **expected invariants**

- Affords **modular** (assume/guarantee) local reasoning

- Analogous to **type checking**:
  - If compiler guarantees that caller passes a string as first argument, callee can assume that first parameter is a string

Ingredient 3: Run-time Support

- Sometimes **compile-time checks** are **not enough**
- Often tools **cannot reason precisely statically** about program behavior
  - Typically **undecidable**!
- Sometimes need to establish, track, and check **global** invariants
  - Global static checking is very expensive and **does not scale to large programs**
- **Design run-time abstractions** to handle difficult cases

`-fbounds-safety` Recipe Instantiation

1. **Local Rule**: Must only dereference pointers with known bounds
2. **Developer Annotations**: Bounds attributes for pointers
3. **Run-time Support**: Codegen to trap if dereference is out of bounds

C++ Safe Buffers Recipe Instantiation

1. **Local Rule**: No raw pointer arithmetic
2. **Developer Annotations**: Adopt `span` utility class for buffers
3. **Run-time Support**: Hardened standard library checks bounds in `span`

WebKit Smart Pointer Safety Recipe Instantiation

1. **Local Rule**: No raw pointers on stack across unknown calls
2. **Developer Annotations**: Adopt `RefPtr` utility class for pointers
3. **Run-time Support**: `RefPtr` itself handles keeping raw pointer alive

Challenge for community:

To what other security problems can we apply this recipe?

Biggest Limiting Factor is Cost of Adoption

- Zero initialization and typed allocator adoption required little adopter work
  - Easy to roll out to hundreds of millions of lines of code
- Bounds safety approaches powerful but required more adoption work
  - Adoption so far in the millions of lines of code

Must invest in tooling to lower adoption cost

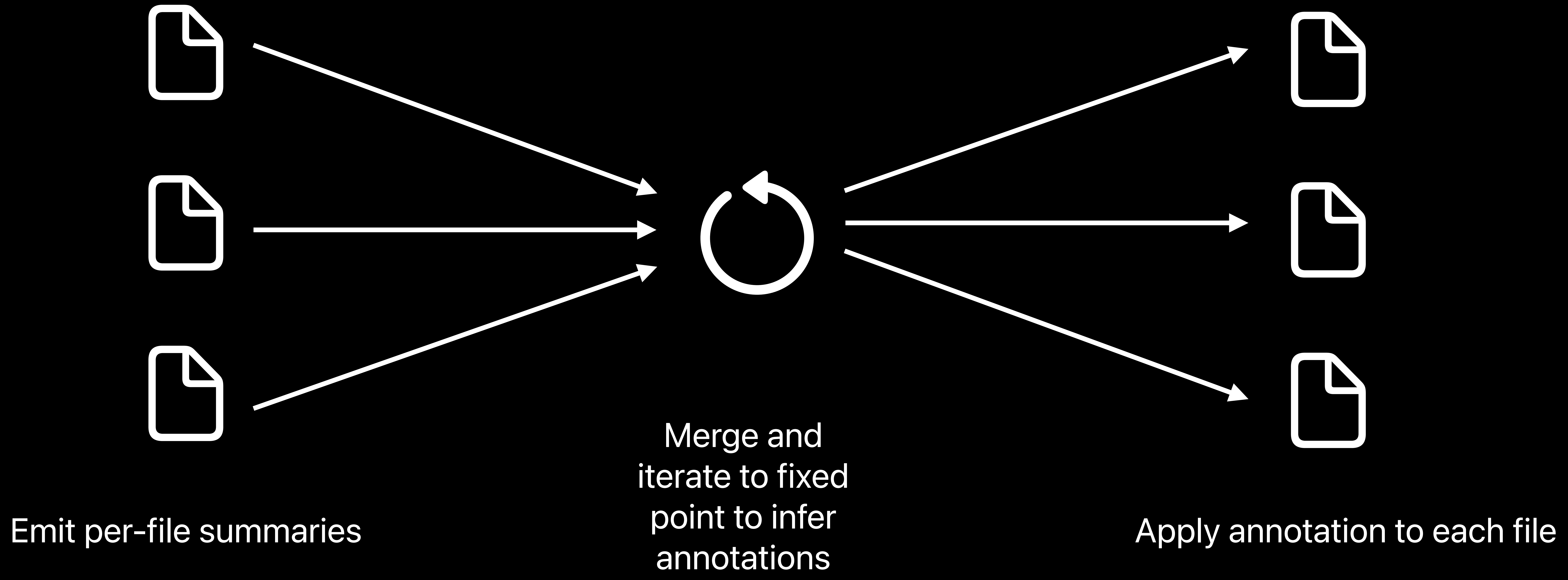
Need Annotation Inference Tooling

- Speed up adoption with by automatically inferring annotations

- Problem:
  - Annotations can be **checked locally,** but must be **inferred globally**
  - E.g., annotations on callee depend on annotations in caller
  - Requires **whole program** reasoning!

Proposal: Summary-Based Inference Tooling

- Achieve whole-program reasoning with **thin-LTO-style summaries**:



Summary Infrastructure Broadly Applicable

- Useful for **Clang Static Analyzer**
  - Enable c**ross translation unit analysis** to **scale to larger codebases**
  - Reduce false positives and false negatives
- Useful for safe C/C++ interoperation with memory-safe languages
  - Infer lifetime annotations
  - Infer escapability annotations

Summary

- Full memory safety only comes with memory safe languages
  - Must support **safe**, **ergonomic interoperation** with memory-safe languages
  - Must **protect ocean of existing C/C++ code** as much as possible

- Strict programming models
  - Recipe for **eliminating** many **classes of memory-safety bugs** in C/C++
  - ❶ Local rules + ❷ Developer-provided annotations + ❸ Run-time support
  - **Annotation cost** is major **limiting factor** in adoption

- Invest in **annotation inference tooling**