-fbounds-safety

Enforcing bounds safety for production C code

Yeoul Na (Apple), May 11th, 2023

Agenda

Motivation

Design goals and highlights

Programming model of -fbounds-safety

Optimization

Performance impact

Memory unsafety is the leading source of security vulnerabilities

Memory unsafety is the leading source of security vulnerabilities

- Memory safety bugs account for 60-70% of software vulnerabilities

Memory unsafety is the leading source of security vulnerabilities

- Memory safety bugs account for 60-70% of software vulnerabilities
- High-profile attacks have exploited memory safety bugs leading to financial and physical threats

Memory safety properties

- Bounds safety (or spatial safety)
- Temporal safety (or lifetime safety)
- Type safety
- Definite initialization
- Thread safety

C does not guarantee memory safety

- ❌ Bounds safety (or spatial safety)
- ❌ Temporal safety (or lifetime safety)
- ❌ Type safety
- ❌ Definite initialization
- ❌ Thread safety

Memory-safe languages provide enhanced safety guarantees

✅ Bounds safety (or spatial safety)

✅ Temporal safety (or lifetime safety)

✅ Type safety

✅ Definite initialization

✅ Thread safety

Memory-safe languages are increasingly the best choice

- Memory-safe languages have emerged as a promising option for systems programming
- Increasingly available for more programming environments
- Incredible initiatives taking place in this domain

Transitioning from C to safe languages takes time

- Billions of lines of C code remain in production
- Efforts to rewrite existing C code using safe languages (e.g., Linux kernel)
- Rewriting requires significant engineering effort and time
- Expect continued maintenance of C code for several more decades

We need a solution to rapidly harden existing C code

Bounds unsafety is the biggest cause of dangerous vulnerabilities

2022 CWE Top 25 Most Dangerous Software Weaknesses

Rank	Name
1	Out-of-bounds Write
4	Improper Input Validation
5	Out-of-bounds Read
13	Integer Overflow or Wraparound
19	Improper Restriction of Operations within the Bounds of a Memory Buffer

Bounds unsafety is the biggest cause of dangerous vulnerabilities

2022 CWE Top 25 Most Dangerous Software Weaknesses

Rank	Name
1	Out-of-bounds Write
4	Improper Input Validation
5	Out-of-bounds Read
13	Integer Overflow or Wraparound
19	Improper Restriction of Operations within the Bounds of a Memory Buffer

Bounds unsafety is the biggest cause of dangerous vulnerabilities

2022 CWE Top 25 Most Dangerous Software Weaknesses

Rank	Name
1	Out-of-bounds Write
4	Improper Input Validation
5	Out-of-bounds Read
13	Integer Overflow or Wraparound
19	Improper Restriction of Operations within the Bounds of a Memory Buffer

Bounds unsafety is the biggest cause of dangerous vulnerabilities

2022 CWE Top 25 Most Dangerous Software Weaknesses

Rank	Name
1	Out-of-bounds Write
4	Improper Input Validation
5	Out-of-bounds Read
13	Integer Overflow or Wraparound
19	Improper Restriction of Operations within the Bounds of a Memory Buffer

-fbounds-safety

C extension for bounds safety

-fbounds-safety only provides bounds safety

But it offers quicker way to make remaining C code safer

	C	-fbounds-safety	Memory safe languages (Swift, Rust, etc.)
Bounds safety (or spatial safety)	❌	✅	✅
Temporal safety	❌	❌	✅
Type safety	❌	❌	✅
Definite initialization	❌	❌	✅
Thread safety	❌	❌	✅

Design goals and highlights

Automatically insert bounds checks as a safety net

- Programmers manually add bounds checks, but sometimes make mistakes
- -fbounds-safety automatically adds bounds checks as a safety net

```c
void fill_array_with_indices(int *buf, size_t count) {
  for (size_t i = 0; i <= count; ++i) {
    buf[i] = i;
  }
}
```

Automatically insert bounds checks as a safety net

- Programmers manually add bounds checks, but sometimes make mistakes
- -fbounds-safety automatically adds bounds checks as a safety net

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

Automatically insert bounds checks as a safety net

- Programmers manually add bounds checks, but sometimes make mistakes
- -fbounds-safety automatically adds bounds checks as a safety net

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

C pointers do not have bounds information

Potential solution: Use wide pointers

Potential solution: Use wide pointers

- Analogous to struct with upper/lower bounds alongside the pointer value

```c
typedef struct {
  int *pointer;
  int *upper_bound;
  int *lower_bound;
} wide_ptr;
```

Potential solution: Use wide pointers

- Analogous to struct with upper/lower bounds alongside the pointer value
- a.k.a “fat” pointers

```c
typedef struct {
  int *pointer;
  int *upper_bound;
  int *lower_bound;
} wide_ptr;
```

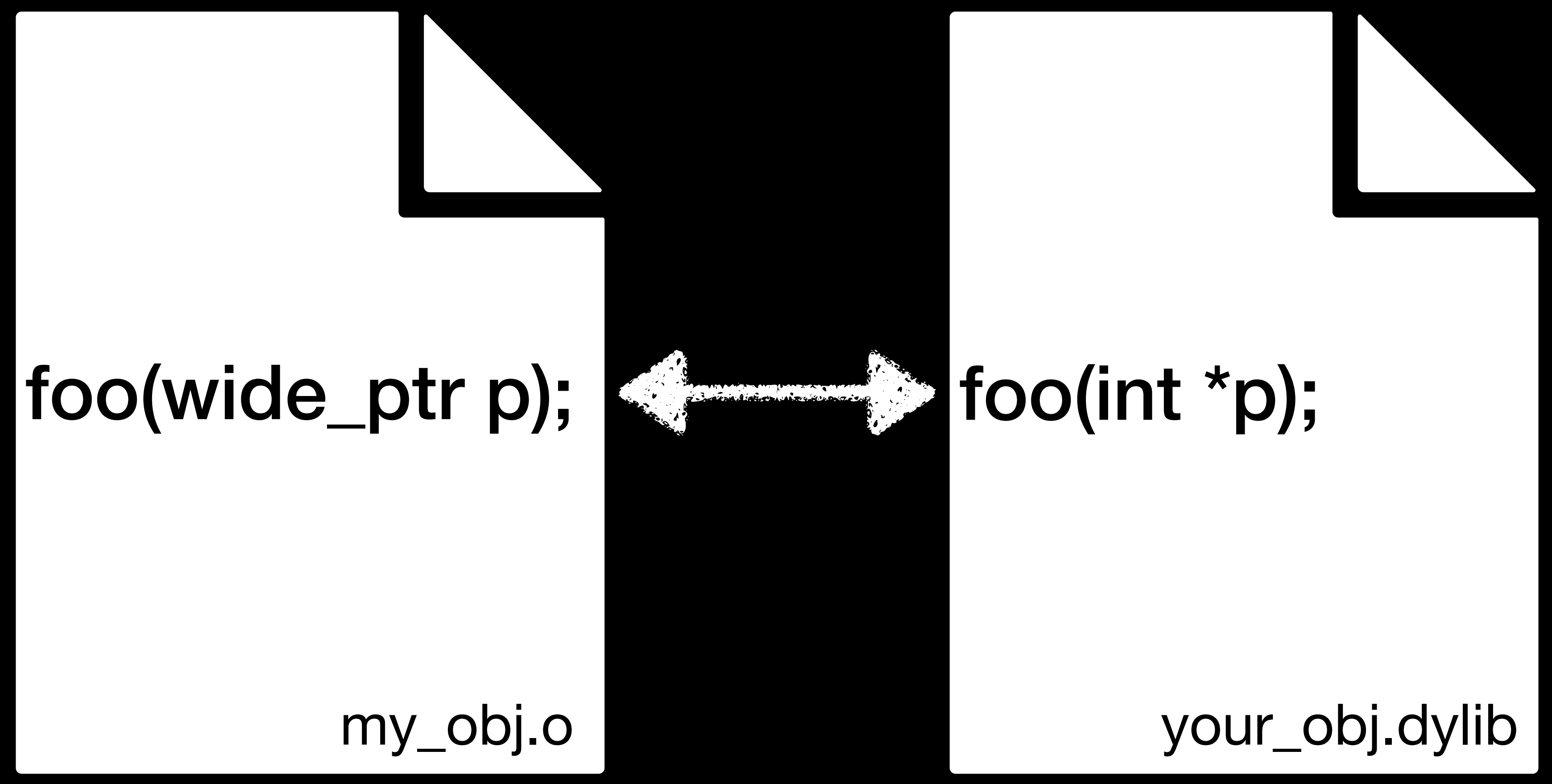
Potential solution: Use wide pointers

- Analogous to struct with upper/lower bounds alongside the pointer value
- a.k.a “fat” pointers
- Allows compiler to automatically insert bounds check

```
typedef struct {
  int *pointer;
  int *upper_bound;
  int *lower_bound;
} wide_ptr;
```

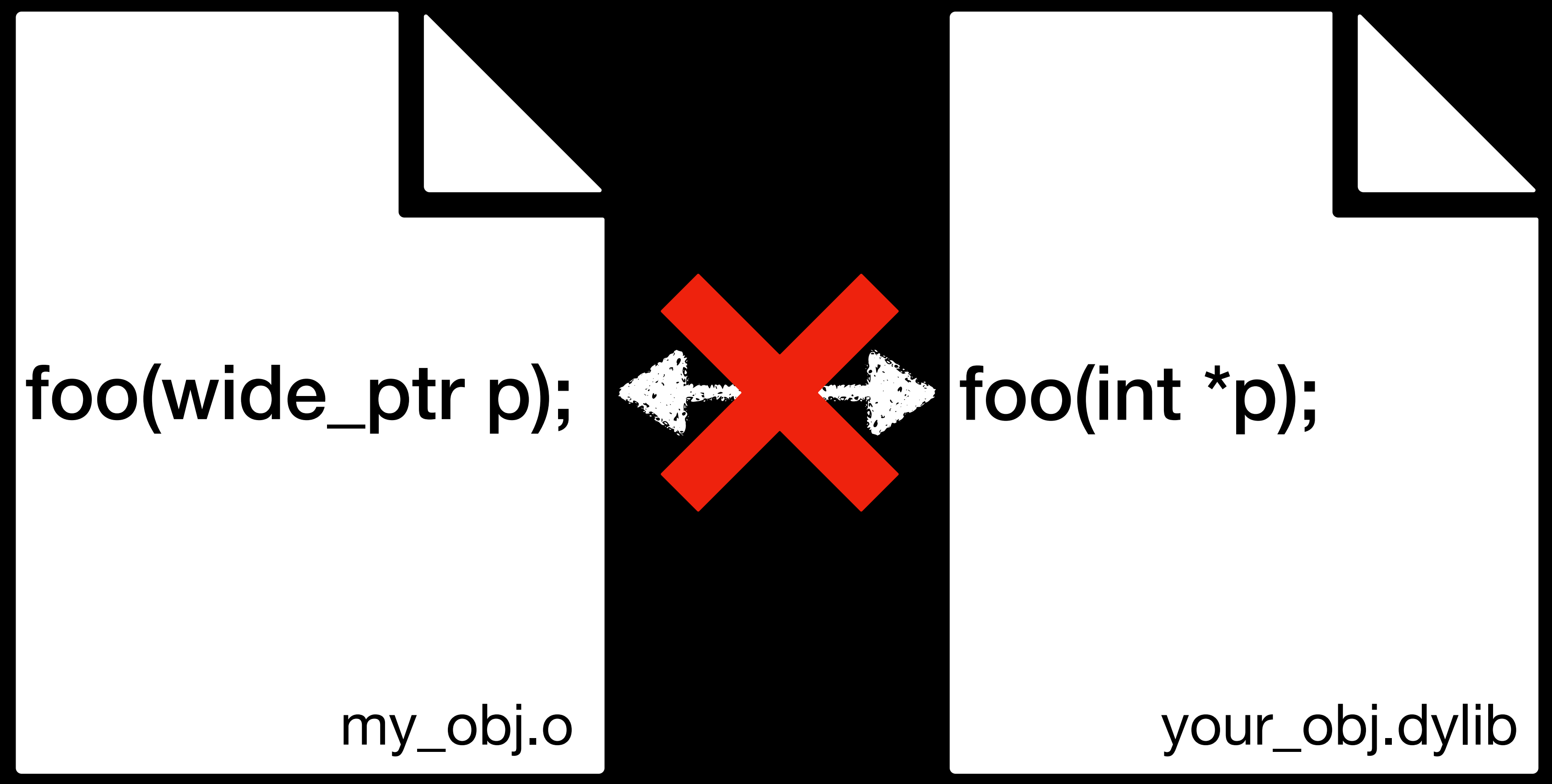
Problem: Wide pointers break Application Binary Interface (ABI)

Problem: Wide pointers break Application Binary Interface (ABI)



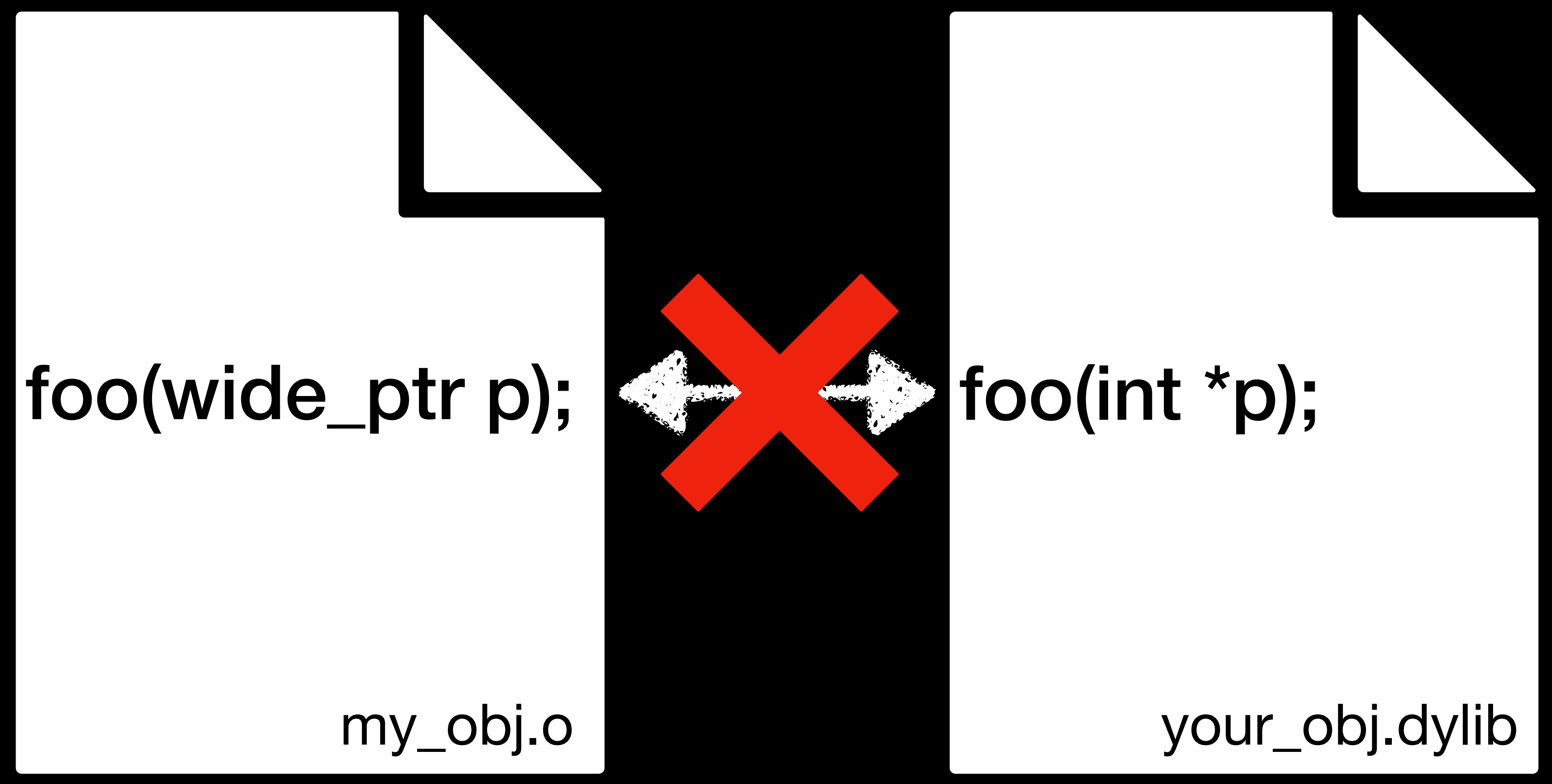
Problem: Wide pointers break Application Binary Interface (ABI)

- Problem interacting with external libraries



Problem: Wide pointers break Application Binary Interface (ABI)

- Problem interacting with external libraries
- Difficult to incrementally adopt the technique



Incremental adoption is crucial

- Adoption often requires significant engineering effort
- Adopting on a large project all at once is likely infeasible

Potential solution: Use bounds annotations

- Require programmers to provide bounds annotation on their code
  - e.g., `void foo(int *__counted_by(n) buf, int n);`
- No need to change pointer representation
- Preserves ABI
- Enables incremental adoption

Potential solution: Use bounds annotations

- Require programmers to provide bounds annotation on their code
  - e.g., `void foo(int *__counted_by(n) buf, int n);`
- No need to change pointer representation
- Preserves ABI
- Enables incremental adoption

Problem: Annotation burden

- Adding annotations on every pointer requires significant programmer effort
- Prevents wide adoption in practice

-fbounds-safety: Mix them together!

-fbounds-safety: Mix them together!

- Wide pointers on non-ABI surface
  - Lowers annotation burden

-fbounds-safety: Mix them together!

- Wide pointers on non-ABI surface
  - Lowers annotation burden
- Bounds annotations on ABI surface
  - Preserves ABI
  - Enables incremental adoption

-fbounds-safety:

Automatic bounds checking with bounds annotations

- Programmers adopt bounds annotations on:
  - Function prototypes, struct fields, globals
- Compiler adds guaranteed bounds checks

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

-fbounds-safety:

Automatic bounds checking with bounds annotations

- Programmers adopt bounds annotations on:
  - Function prototypes, struct fields, globals
- Compiler adds guaranteed bounds checks

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

-fbounds-safety:
Automatic bounds checking with bounds annotations

- Programmers adopt bounds annotations on:
  - Function prototypes, struct fields, globals
- Compiler adds guaranteed bounds checks

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

Compiler rejects code without sufficient bounds information

- Guides programmers to add necessary bounds annotations
- Securing all pointers by default

Compiler rejects code without sufficient bounds information

- Guides programmers to add necessary bounds annotations
- Securing all pointers by default

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

Compiler rejects code without sufficient bounds information

- Guides programmers to add necessary bounds annotations
- Securing all pointers by default

```c
void fill_array_with_indices(int *buf, size_t count) {
  for (size_t i = 0; i <= count; ++i) {
    buf[i] = i;
  }
}
```

Array subscript on single pointer ‘buf’ must use a constant index of 0 to be in bounds

Compiler rejects code without sufficient bounds information

- Guides programmers to add necessary bounds annotations
- Securing all pointers by default

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

-fbounds-safety doesn’t require bounds annotations all the time

Local variables are wide by default

Solution to keep bounds annotation burden low

- Compiler implicitly carries bounds for local variables
  - No manual annotation is required
- No ABI implications

```
void foo(int i) {
  char *buf = (char *)malloc(10);
  if (buf + i < buf || buf + i >= buf + 10) trap(); // automatically inserted
  buf[i] = 0xff;
  // more code ...
}
```

Local variables are wide by default

Solution to keep bounds annotation burden low

- Compiler implicitly carries bounds for local variables
  - No manual annotation is required
- No ABI implications

```
void foo(int i) {
  char *buf = (char *)malloc(10);
  if (buf + i < buf || buf + i >= buf + 10) trap(); // automatically inserted
  buf[i] = 0xff;
  // more code ...
}
```

Local variables are wide by default

Solution to keep bounds annotation burden low

- Compiler implicitly carries bounds for local variables
  - No manual annotation is required
- No ABI implications

```c
void foo(int i) {
  char *buf = (char *)malloc(10);
  if (buf + i < buf || buf + i >= buf + 10) trap(); // automatically inserted
  buf[i] = 0xff;
  // more code ...
}
```

All pointers except locals are single by default

- Most pointers are pointing to a single object
  - No need for pointer arithmetic
  - No need for bounds information
- Annotation __single is default for all pointers except locals

```
void fill_struct(struct s_t *p);

// example usage
struct s_t s;
fill_struct(&s);
```

All pointers except locals are single by default

- Most pointers are pointing to a single object
  - No need for pointer arithmetic
  - No need for bounds information
- Annotation __single is default for all pointers except locals

```
void fill_struct(struct s_t *__single p);

// example usage
struct s_t s;
fill_struct(&s);
```

All pointers except locals are single by default

- Most pointers are pointing to a single object
  - No need for pointer arithmetic
  - No need for bounds information
- Annotation __single is default for all pointers except locals

```
void fill_struct(struct s_t *p);

// example usage
struct s_t s;
fill_struct(&s);
```

-fbounds-safety solves challenges for safe C extensions

-fbounds-safety solves challenges for safe C extensions

- ABI compatibility ✅
- Incremental adoption ✅
- Adoption burden ✅

-fbounds-safety solves challenges for safe C extensions

- ABI compatibility ✅
- Incremental adoption ✅
- Adoption burden ✅
- Source compatibility ?

Challenge: Preserve source compatibility with C

Challenge: Preserve source compatibility with C

- Need to build with standard C compilers

Challenge: Preserve source compatibility with C

- Need to build with standard C compilers
- Need compatibility with existing static analysis and code inspection tooling

Challenge: Preserve source compatibility with C

- Need to build with standard C compilers
- Need compatibility with existing static analysis and code inspection tooling
- Should be adoptable in shared code:

Challenge: Preserve source compatibility with C

- Need to build with standard C compilers
- Need compatibility with existing static analysis and code inspection tooling
- Should be adoptable in shared code:
  - Library headers

Challenge: Preserve source compatibility with C

- Need to build with standard C compilers
- Need compatibility with existing static analysis and code inspection tooling
- Should be adoptable in shared code:
  - Library headers
  - Open-source projects

Challenge: Preserve source compatibility with C

- Need to build with standard C compilers
- Need compatibility with existing static analysis and code inspection tooling
- Should be adoptable in shared code:
  - Library headers
  - Open-source projects
- **Requirement: Must not introduce new syntax that C compilers don't understand**

Bounds annotations are macro-defined C attributes

- Bounds annotations are syntactically C type attributes
  - Do not introduce new syntax
- Bounds annotations are macro-defined
  - When defined to empty they are still valid C

-fbounds-safety solves real-world challenges

- ABI compatibility ✅
- Incremental adoption ✅
- Low adoption burden ✅
- Source compatibility ?

-fbounds-safety solves real-world challenges

- ABI compatibility ✅
- Incremental adoption ✅
- Low adoption burden ✅
- Source compatibility ✅

-fbounds-safety is relatively easy to adopt

-fbounds-safety is relatively easy to adopt

- Adoption is mostly about annotations in function prototypes and struct fields

-fbounds-safety is relatively easy to adopt

- Adoption is mostly about annotations in function prototypes and struct fields
- Require relatively less code modification

-fbounds-safety is relatively easy to adopt

- Adoption is mostly about annotations in function prototypes and struct fields
- Require relatively less code modification
- Time to adopt: ~ 1 hour per 2,000 LOC (vary depending on codebase)

-fbounds-safety is relatively easy to adopt

- Adoption is mostly about annotations in function prototypes and struct fields
- Require relatively less code modification
- Time to adopt: ~ 1 hour per 2,000 LOC (vary depending on codebase)
- Currently, only supports C (Objective-C and C++ are not supported)

-fbounds-safety is relatively easy to adopt

- Adoption is mostly about annotations in function prototypes and struct fields
- Require relatively less code modification
- Time to adopt: ~ 1 hour per 2,000 LOC (vary depending on codebase)
- Currently, only supports C (Objective-C and C++ are not supported)
- Can mix and match bounds safe and unsafe code

-fbounds-safety is relatively easy to adopt

- Adoption is mostly about annotations in function prototypes and struct fields
- Require relatively less code modification
- Time to adopt: ~ 1 hour per 2,000 LOC (vary depending on codebase)
- Currently, only supports C (Objective-C and C++ are not supported)
- Can mix and match bounds safe and unsafe code
- Allows strategy of incremental adoption

Adoption at Apple

Adoption at Apple

- Adopted in millions of lines of production C code

Adoption at Apple

- Adopted in millions of lines of production C code
- Libraries used for:

Adoption at Apple

- Adopted in millions of lines of production C code
- Libraries used for:
  - Secure boot and firmware

Adoption at Apple

- Adopted in millions of lines of production C code
- Libraries used for:
  - Secure boot and firmware
  - Security-critical components of XNU

Adoption at Apple

- Adopted in millions of lines of production C code
- Libraries used for:
    - Secure boot and firmware
    - Security-critical components of XNU
    - Built-in image format parsers

Adoption at Apple

- Adopted in millions of lines of production C code
- Libraries used for:
    - Secure boot and firmware
    - Security-critical components of XNU
    - Built-in image format parsers
    - Built-in audio codecs

Adoption at Apple

- Adopted in millions of lines of production C code
- Libraries used for:
  - Secure boot and firmware
  - Security-critical components of XNU
  - Built-in image format parsers
  - Built-in audio codecs
- Found to be effective for real-world applications

Programming model

Enforcing bounds safety at language level

Bounds annotations

-fbounds-safety enforces bounds safety at language level

-fbounds-safety enforces bounds safety at language level

- Prevents out-of-bounds memory accesses via bounds checking

-fbounds-safety enforces bounds safety at language level

- Prevents out-of-bounds memory accesses via bounds checking
- Prevents pointer operations that cannot be proven safe (or with insufficient bounds information)

-fbounds-safety enforces bounds safety at language level

- Prevents out-of-bounds memory accesses via bounds checking
- Prevents pointer operations that cannot be proven safe (or with insufficient bounds information)
- Maintains correctness of bounds annotations

-fbounds-safety prevents unsafe behaviors by …

- Compile-time warning / error when the compiler knows an operation will be unsafe
- Run-time checks and traps when behavior cannot be proven safe/unsafe at compile time
- Compiler uses its best effort to report errors at compile time

Bounds annotations

External bounds annotations

Describe relationship between pointer and bounds information

`count` is the element count of `buf`

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

Bounds annotation: __counted_by(N)

- `buf` has `count` elements with the valid range [0, count)
- Can be indexed in a positive direction ➡️
- Can be used inside array bracket,.e.g, int arr[__counted_by(count)]

```
void fill_array_int(int *__counted_by(count) buf, size_t count);

// example usage
fill_array_int(array, 10);
```

Bounds annotation: __sized_by(N)

- `buf` has `byte_count` with the valid range [0, byte_count)
- Can be indexed in a positive direction ➡️

```
void fill_array_byte(void *__sized_by(byte_count) buf, size_t byte_count);

// example usage
fill_array_byte(array, 10 * sizeof(array[0]));
```

Bounds annotation: __ended_by(P)

- `end` is the upper bound of `buf` with the valid range [0, end - buf)
- `buf` indexed in a positive direction ➡️; `end` in a negative direction ⬅️

```
void fill_array_to_end(int *__ended_by(end) buf, int *end);

// example usage
fill_array_to_end(array, &array[10]);
```

Maintaining correctness of __counted_by

Updating pointer may invalidate bounds information

```c
void foo(int *__counted_by(count) buf, size_t count) {
  buf = (int *)malloc(4);
}
// usage
foo(buf , 10);
```

Updating pointer may invalidate bounds information

- Updating `buf` to point to an object of byte size 4

```
void foo(int *__counted_by(count) buf, size_t count) {
  buf = (int *)malloc(4);
}
// usage
foo(buf , 10);
```

Updating pointer may invalidate bounds information

- Updating `buf` to point to an object of byte size 4
- The count variable is 10 so __count_by annotation becomes invalid

```
void foo(int *__counted_by(count) buf, size_t count) {
  buf = (int *)malloc(4);
}
// usage
foo(buf , 10);
```

Updating pointer may invalidate bounds information

- Updating `buf` to point to an object of byte size 4
- The count variable is 10 so __count_by annotation becomes invalid

```
void foo(int *__counted_by(count) buf, size_t count) {
  buf = (int *)malloc(4);
}
// usage
foo(buf , 10);
```

Assignment to 'int * __counted_by(count)' 'buf' requires corresponding assignment to 'count'

Updating pointer may invalidate bounds information

- Updating `buf` to point to an object of byte size 4
- The count variable is 10 so __count_by annotation becomes invalid

```
void foo(int *__counted_by(count) buf, size_t count) {
  buf = (int *)malloc(4);
  count = 4;
}
// usage
foo(buf , 10);
```

Updating pointer may invalidate bounds information

- Updating `buf` to point to an object of byte size 4
- The count variable is 10 so __count_by annotation becomes invalid

```
void foo(int *__counted_by(count) buf, size_t count) {
  if (4 * sizeof(int) > 4) trap();
  buf = (int *)malloc(4);
  count = 4;
}
// usage
foo(buf , 10);
```

Annotation for C strings: __null_terminated

```c
size_t my_strlen(const char *__null_terminated str);

// example usage
size_t ak_len = my_strlen(“abcdefghijk”);
```

Annotation for C strings: __null_terminated

- Indicates `str` has the null terminator

```
size_t my_strlen(const char *__null_terminated str);

// example usage
size_t ak_len = my_strlen("abcdefghijk");
```

Annotation for C strings: __null_terminated

- Indicates `str` has the null terminator
- Ensures that `str` is not accessed beyond the null terminator

```c
size_t my_strlen(const char *__null_terminated str);

// example usage
size_t ak_len = my_strlen("abcdefghijk");
```

__single: pointers to single object

- `p` is pointing to a single valid object (this is the case for most pointers)
- Can NOT be indexed in any direction 🚫

```
void fill_struct(struct s_t *__single p);

// example usage
struct s_t s = {};
fill_struct(&s);
```

Help us support more use cases with your feedback!

Internal bounds annotations

Escape hatches that allow to explicitly use wide pointers

Internal bounds annotation: __bidi_indexable

```c
void fill_array_internal_bounds(int *__bidi_indexable buf);

// example usage
int array[10] = {0};
fill_array_internal_bounds(array);
```

Internal bounds annotation: __bidi_indexable

- __bidi_indexable turns `buf` into a wide pointer with upper and lower bounds

```
void fill_array_internal_bounds(int *__bidi_indexable buf);

// example usage
int array[10] = {0};
fill_array_internal_bounds(array);
```

Internal bounds annotation: __bidi_indexable

- __bidi_indexable turns `buf` into a wide pointer with upper and lower bounds
- Can be indexed in both directions ↔️

```
void fill_array_internal_bounds(int *__bidi_indexable buf);

// example usage
int array[10] = {0};
fill_array_internal_bounds(array);
```

Internal bounds annotation: __bidi_indexable

- __bidi_indexable turns `buf` into a wide pointer with upper and lower bounds
- Can be indexed in both directions ↔️
- Changes the pointer representation -> breaks the ABI

```
void fill_array_internal_bounds(int *__bidi_indexable buf);

// example usage
int array[10] = {0};
fill_array_internal_bounds(array);
```

Internal bounds annotation: __bidi_indexable

- __bidi_indexable turns `buf` into a wide pointer with upper and lower bounds
- Can be indexed in both directions ↔️
- Changes the pointer representation -> breaks the ABI
- ⚠️ Avoid using it on the ABI surface

```
void fill_array_internal_bounds(int *__bidi_indexable buf);

// example usage
int array[10] = {0};
fill_array_internal_bounds(array);
```

Internal bounds annotation: __indexable

```
void fill_array_internal_bounds(int *__indexable buf);

// example usage
int array[10] = {0};
fill_array_internal_bounds(array);
```

Internal bounds annotation: __indexable

- `buf` is a wide pointer with upper bound (smaller than __bidi_indexable)

```
void fill_array_internal_bounds(int *__indexable buf);

// example usage
int array[10] = {0};
fill_array_internal_bounds(array);
```

Internal bounds annotation: __indexable

- `buf` is a wide pointer with upper bound (smaller than __bidi_indexable)
- Can be indexed in a positive direction ➡️

```
void fill_array_internal_bounds(int *__indexable buf);

// example usage
int array[10] = {0};
fill_array_internal_bounds(array);
```

Internal bounds annotation: __indexable

- `buf` is a wide pointer with upper bound (smaller than __bidi_indexable)
- Can be indexed in a positive direction ➡️
- Changes the pointer representation -> breaks ABI

```
void fill_array_internal_bounds(int *__indexable buf);

// example usage
int array[10] = {0};
fill_array_internal_bounds(array);
```

Internal bounds annotation: __indexable

- `buf` is a wide pointer with upper bound (smaller than __bidi_indexable)
- Can be indexed in a positive direction ➡️
- Changes the pointer representation -> breaks ABI
- ⚠️ Avoid using it on the ABI surface

```c
void fill_array_internal_bounds(int *__indexable buf);

// example usage
int array[10] = {0};
fill_array_internal_bounds(array);
```

Default pointer annotations

Key for ABI compatibility & less manual annotation

- ABI visible pointers : __single by default
- Non-ABI visible pointers : __bidi_indexable by default
- const char * : __null_terminated by default

- Secures all pointers by default
- Preserves ABI compatibility by default
- Doesn’t need manual annotation all the time

Interoperability w/ bounds-unsafe code enables incremental adoption

__unsafe_indexable: pointers from bounds-unsafe code

__unsafe_indexable: pointers from bounds-unsafe code

- Just like normal C pointers
  - Can be indexed in both directions ↔️
  - No checks are added ❌

```
// my_system.h
void *__unsafe_indexable system_function(void *__unsafe_indexable buf);
```

__unsafe_indexable: pointers from bounds-unsafe code

- Just like normal C pointers
  - Can be indexed in both directions ↔️
  - No checks are added ❌
- Avoid using in bounds-safe code

```
// my_system.h
void *__unsafe_indexable system_function(void *__unsafe_indexable buf);
```

__unsafe_indexable: pointers from bounds-unsafe code

- Just like normal C pointers
  - Can be indexed in both directions ↔️
  - No checks are added ❌
- Avoid using in bounds-safe code
- Default for system headers

```
// my_system.h
void *__unsafe_indexable system_function(void *__unsafe_indexable buf);
```

Taking a return value from unsafe code

Taking a return value from unsafe code

- The model doesn't allow initializing any safe pointer with an unsafe pointer

```
int *safe_buf = unsafe_func(); // error
```

Taking a return value from unsafe code

- The model doesn't allow initializing any safe pointer with an unsafe pointer

```
int *safe_buf = unsafe_func(); // error
```

- Use __unsafe_forge_bidi_indexable (T,P,S) to create a __bidi_indexable pointer from an __unsafe_indexable pointer

```
int *safe_buf =
  __unsafe_forge_bidi_indexable(int *, unsafe_func(), byte_size_of_buf); // ok
```

Taking a return value from unsafe code

- The model doesn't allow initializing any safe pointer with an unsafe pointer

```
int *safe_buf = unsafe_func(); // error
```

- Use __unsafe_forge_bidi_indexable (T,P,S) to create a __bidi_indexable pointer from an __unsafe_indexable pointer

```
int *safe_buf =
  __unsafe_forge_bidi_indexable(int *, unsafe_func(), byte_size_of_buf); // ok
```

Avoid using this intrinsic for any other purposes!

Bounds annotation summary

	ABI Compatibility	Index directions	Bounds checks	Default for
__counted_by(N)	✅	➡️	✅	
__sized_by(N)	✅	➡️	✅	
__ended_by(N)	✅	➡️	✅	
__null_terminated	✅	1️⃣➡️	✅	const char *
__single	✅	❌	✅	Function prototypes / struct fields / globals
__indexable	❌ (2x bigger)	➡️	✅	
__bidi_indexable	❌ (3x bigger)	↔️	✅	Locals
__unsafe_indexable	✅	↔️	❌	Pointers in system headers

Bounds annotation summary

	ABI Compatibility	Index directions	Bounds checks	Default for
__counted_by(N)	✅	➡️	✅	
__sized_by(N)	✅	➡️	✅	
__ended_by(N)	✅	➡️	✅	
__null_terminated	✅	1️⃣➡️	✅	const char *
__single	✅	❌	✅	Function prototypes / struct fields / globals
__indexable	❌ (2x bigger)	➡️	✅	
__bidi_indexable	❌ (3x bigger)	↔️	✅	Locals
__unsafe_indexable	✅	↔️	❌	Pointers in system headers

Bounds annotation summary

	ABI Compatibility	Index directions	Bounds checks	Default for
__counted_by(N)	✅	➡️	✅	
__sized_by(N)	✅	➡️	✅	
__ended_by(N)	✅	➡️	✅	
__null_terminated	✅	1️⃣➡️	✅	const char *
__single	✅	❌	✅	Function prototypes / struct fields / globals
__indexable	❌ (2x bigger)	➡️	✅	
__bidi_indexable	❌ (3x bigger)	↔️	✅	Locals
__unsafe_indexable	✅	↔️	❌	Pointers in system headers

Optimization to remove redundant bounds checks

Optimization to remove redundant run-time checks

Optimization to remove redundant run-time checks

- Automatic bounds checks may introduce redundant checks

Optimization to remove redundant run-time checks

- Automatic bounds checks may introduce redundant checks

```
for (size_t i = 0; i < count; ++i) {
  if (i < 0 || i >= count) trap(); // automatically added bounds checks
  buf[i] = i;
}
```

Optimization to remove redundant run-time checks

- Automatic bounds checks may introduce redundant checks

```
for (size_t i = 0; i < count; ++i) {
  if (i < 0 || i >= count) trap(); // automatically added bounds checks
  buf[i] = i;
}
```

Optimization to remove redundant run-time checks

- Automatic bounds checks may introduce redundant checks

```
for (size_t i = 0; i < count; ++i) {
  if (i < 0 || i >= count) trap(); // automatically added bounds checks
  buf[i] = i;
}
```

Optimization to remove redundant run-time checks

- Automatic bounds checks may introduce redundant checks
- LLVM optimizer remove redundant checks

```
for (size_t i = 0; i < count; ++i) {
  if (i < 0 || i >= count) trap(); // automatically added bounds checks
  buf[i] = i;
}
```

Optimization to remove redundant run-time checks

- Automatic bounds checks may introduce redundant checks
- LLVM optimizer remove redundant checks
- Primary motivation for the constraint-elimination pass we implement in LLVM

```
for (size_t i = 0; i < count; ++i) {
  if (i < 0 || i >= count) trap(); // automatically added bounds checks
  buf[i] = i;
}
```

Constraint elimination to remove redundant checks

- Collect known conditions through CFG
- Remove redundant checks based on the known conditions

```
for (size_t i = 0; i < count; ++i) {
   // known fact: 0 <= i < count
   if (i < 0 || i >= count) trap(); // <- always `false`
   buf[i] = i;
}
```

- Fewer checks will be inserted if the code already has most of the necessary bounds checks

Constraint elimination to remove redundant checks

- Collect known conditions through CFG
- Remove redundant checks based on the known conditions

```
 for (size_t i = 0; i < count; ++i) {
    // known fact: 0 <= i < count
    if (i < 0 || i >= count) trap(); // <- always `false`
    buf[i] = i;
}
```

- Fewer checks will be inserted if the code already has most of the necessary bounds checks

Constraint elimination to remove redundant checks

- Collect known conditions through CFG
- Remove redundant checks based on the known conditions

```
for (size_t i = 0; i < count; ++i) {
    // known fact: 0 <= i < count
    if (i < 0 || i >= count) trap(); // <- always `false`
    buf[i] = i;
}
```

- Fewer checks will be inserted if the code already has most of the necessary bounds checks

Constraint elimination to remove redundant checks

- Collect known conditions through CFG
- Remove redundant checks based on the known conditions

```
for (size_t i = 0; i < count; ++i) {
    // known fact: 0 <= i < count
    if (i < 0 || i >= count) trap(); // <- always `false`
    buf[i] = i;
}
```

- Fewer checks will be inserted if the code already has most of the necessary bounds checks

Constraint elimination to remove redundant checks

- Collect known conditions through CFG
- Remove redundant checks based on the known conditions

```
 for (size_t i = 0; i < count; ++i) {
    // known fact: 0 <= i < count
    if (0) trap(); // <- always `false`
    buf[i] = i;
}
```

- Fewer checks will be inserted if the code already has most of the necessary bounds checks

Constraint elimination to remove redundant checks

- Collect known conditions through CFG
- Remove redundant checks based on the known conditions

```
for (size_t i = 0; i < count; ++i) {
    // known fact: 0 <= i < count

    buf[i] = i;
}
```

- Fewer checks will be inserted if the code already has most of the necessary bounds checks

Performance impact

Benchmark results
w/ Ptrdist and Olden benchmark suites

- Pointer-intensive benchmark suites used by other related approaches
- Did not adopt two benchmarks, one in Ptrdist and one in Olden

Benchmark results

w/ Ptrdist and Olden benchmark suites

Benchmark results

w/ Ptrdist and Olden benchmark suites

- LOC changes: 2.7% (0.2% used unsafe constructs)

Benchmark results

w/ Ptrdist and Olden benchmark suites

- LOC changes: 2.7% (0.2% used unsafe constructs)
  - Much lower than prior approaches

Benchmark results

w/ Ptrdist and Olden benchmark suites

- LOC changes: 2.7% (0.2% used unsafe constructs)
  - Much lower than prior approaches
- Compile-time overhead: 11%

Benchmark results

w/ Ptrdist and Olden benchmark suites

- LOC changes: 2.7% (0.2% used unsafe constructs)
  - Much lower than prior approaches
- Compile-time overhead: 11%
- Code-size (text section) overhead: 9.1% (ranged -1.4% to 38%)

Benchmark results

w/ Ptrdist and Olden benchmark suites

- LOC changes: 2.7% (0.2% used unsafe constructs)
  - Much lower than prior approaches
- Compile-time overhead: 11%
- Code-size (text section) overhead: 9.1% (ranged -1.4% to 38%)
- Run-time overhead: 5.1% (ranged -1% to 29%)

Benchmark results
w/ Ptrdist and Olden benchmark suites

- LOC changes: 2.7% (0.2% used unsafe constructs)
  - Much lower than prior approaches
- Compile-time overhead: 11%
- Code-size (text section) overhead: 9.1% (ranged -1.4% to 38%)
- Run-time overhead: 5.1% (ranged -1% to 29%)
  - Tend to rely more on run-time checks with benefit of lower adoption cost

Benchmark results

w/ Ptrdist and Olden benchmark suites

- LOC changes: 2.7% (0.2% used unsafe constructs)
  - Much lower than prior approaches
- Compile-time overhead: 11%
- Code-size (text section) overhead: 9.1% (ranged -1.4% to 38%)
- Run-time overhead: 5.1% (ranged -1% to 29%)
  - Tend to rely more on run-time checks with benefit of lower adoption cost
  - Can be improved with optimization improvements

System-level performance impact

- Measurement on iOS
- 0-8% binary size increase per project
- No measurable performance or power impact on boot, app launch
- Minor overall performance impact on audio decoding/encoding (1%)

- **System-level performance cost is remarkably low and worth paying for the security benefit**

Acknowledgments

- Félix Cloutier
- Patryk Stefanski
- Dan Liew
- Henrik Olsson
- Florian Hahn
- Devin Coughlin
- Filip Pizlo

-fbounds-safety is coming to LLVM community

-fbounds-safety is coming to LLVM community

- -fbounds-safety is a bounds safe C extension widely adopted in shipping software running on all Apple platforms, offering:

-fbounds-safety is coming to LLVM community

- -fbounds-safety is a bounds safe C extension widely adopted in shipping software running on all Apple platforms, offering:
    - ABI compatibility

-fbounds-safety is coming to LLVM community

- -fbounds-safety is a bounds safe C extension widely adopted in shipping software running on all Apple platforms, offering:
  - ABI compatibility
  - Incremental adoption

-fbounds-safety is coming to LLVM community

- -fbounds-safety is a bounds safe C extension widely adopted in shipping software running on all Apple platforms, offering:
  - ABI compatibility
  - Incremental adoption
  - Moderate annotation burden

-fbounds-safety is coming to LLVM community

- -fbounds-safety is a bounds safe C extension widely adopted in shipping software running on all Apple platforms, offering:
    - ABI compatibility
    - Incremental adoption
    - Moderate annotation burden
    - Source compatibility with C

-fbounds-safety is coming to LLVM community

- -fbounds-safety is a bounds safe C extension widely adopted in shipping software running on all Apple platforms, offering:
  - ABI compatibility
  - Incremental adoption
  - Moderate annotation burden
  - Source compatibility with C
- Planning to upstream and standardize the language model

-fbounds-safety is coming to LLVM community

- -fbounds-safety is a bounds safe C extension widely adopted in shipping software running on all Apple platforms, offering:
  - ABI compatibility
  - Incremental adoption
  - Moderate annotation burden
  - Source compatibility with C
- Planning to upstream and standardize the language model
- RFC is coming soon — we are very excited to get your feedback!