Adopting -fbounds-safety in practice

Henrik Olsson & Patryk Stefanski

EuroLLVM 2025

Agenda

Introduction to -fbounds-safety
Live adoption
Live debugging

Current status

- Announced in 2023
- In the process of upstreaming into mainline Clang
- Our goal is to standardize -fbounds-safety in C
- Implementation now open sourced in Swift's fork of Clang
  - https://github.com/swiftlang/llvm-project/
- You can try it out today with the swift.org snapshot toolchain (not in Xcode)
  - https://www.swift.org/install
- Currently only supports C (Objective-C and C++ not yet supported)

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
- Many security critical codebases are written in C
- Full rewrites are costly and time consuming

-fbounds-safety prevents out of bounds memory access

- Mitigations such as FORTIFY_SOURCE do not catch all OOB accesses
- -fbounds-safety provides a strong bounds safety guarantee
- OOB access bugs become unexploitable
- Attackers are forced to look for other types of bugs that are harder to exploit
- Dynamic bug finding tools like ASan don’t protect in production

How bounds safety is guaranteed

-fbounds-safety enforces bounds safety at language level

- Prevents out-of-bounds memory accesses via bounds checking at run time
- Prevents pointer operations at compile time if they cannot be proven safe
- Bounds information can be provided using bounds annotations
- Maintains correctness of bounds annotations

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;
  }
}
```

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;
  }
}
```

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;
  }
}
```

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

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

Array subscript not allowed on pointer without bounds information

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;
  }
}
```

Compiler maintains correctness of bounds annotations

- Pointer and bounds must be kept in sync
- This guarantees correctness of bounds checks

Compiler maintains correctness of bounds annotations

- Pointer and bounds must be kept in sync
- This guarantees correctness of bounds checks

```
void fill_array_with_indices_inverse(int *__counted_by(count) buf, size_t count) {
  while (count-- > 0) {
    *buf = count;
    buf++;
  }
}
```

Compiler maintains correctness of bounds annotations

- Pointer and bounds must be kept in sync
- This guarantees correctness of bounds checks

```
void fill_array_with_indices_inverse(int *__counted_by(count) buf, size_t count) {
  while (count-- > 0) {
    *buf = count;
    buf++;
  }
}
```

Assignment to ‘buf’ requires corresponding assignment to ‘count’

Compiler maintains correctness of bounds annotations

- Pointer and bounds must be kept in sync
- This guarantees correctness of bounds checks

```
void fill_array_with_indices_inverse(int *__counted_by(count) buf, size_t count) {
  while (count > 0) {
    *buf = count - 1;
    buf++;
    count--;
  }
}
```

Local variables track bounds without annotations

- Reduces the number of annotations needed drastically
- Allows for flexibility with reassignments

Local variables track bounds without annotations

- Reduces the number of annotations needed drastically
- Allows for flexibility with reassignments

```
void fill_array_with_indices_inverse(int *__counted_by(count) buf,
                                     size_t count) {
  while (count-- > 0) {
    *buf = count;
    buf++;
  }
}
```

Assignment to ‘buf’ requires corresponding assignment to ‘count’

Local variables track bounds without annotations

- Reduces the number of annotations needed drastically
- Allows for flexibility with reassignments

```
void fill_array_with_indices_inverse(int *__counted_by(countOrig) bufOrig,
                                     size_t countOrig) {
  int *buf = bufOrig;
  size_t count = countOrig;
  while (count-- > 0) {
    *buf = count;
    buf++;
  }
}
```

-fbounds-safety is easy to adopt

- Low manual annotation overhead
- Time to adopt: ~1 hour per 2,000 LOC
- Maintains ABI compatibility
- Allows incremental adoption

Adoption at Apple

- Adopted in millions of lines of production C code
- Libraries used for:
  - Secure boot and firmware
  - Security-critical components of XNU
    - https://github.com/apple-oss-distributions/xnu
  - Built-in image format parsers
  - Built-in audio codecs
- Works well with real-world applications
- Low system level overhead

Performance

Performance

- Note: measured in 2023

Performance

- Note: measured in 2023
- Ptrdist and Olden benchmark suites
  - Code size overhead (text section only): 9.1% geomean (range: -1.4% to 38%)
  - Run-time overhead: 5.1% geomean (range: -1% to 29%)
    - Can be optimized further

Performance

- Note: measured in 2023
- Ptrdist and Olden benchmark suites
  - Code size overhead (text section only): 9.1% geomean (range: -1.4% to 38%)
  - Run-time overhead: 5.1% geomean (range: -1% to 29%)
    - Can be optimized further
- Minor run-time impact on real-world adopters
  - audio encoding/decoding: ~1% overhead

Bounds annotations

__counted_by carries bounds info across interfaces

- Let the programmer specify where the size is stored

```
void foo(int *__counted_by(len) p, size_t len);

void bar(int *__counted_by(42) q);

void baz(int *__counted_by(a * b) p, size_t a, size_t b);
```

- No need to change the pointer representation (preserves ABI)
- Compile-time and run-time checks to enforce that pointer and count are in sync

__counted_by variants for different use cases

```
void bzero(void *__counted_by(n) s, size_t n);
```

__counted_by variants for different use cases

```
void bzero(void *__sized_by(n) s, size_t n);
```

- __sized_by(size) — size denotes the size in bytes instead of number of elements

__counted_by variants for different use cases

```
void bzero(void *__sized_by(n) s, size_t n);
```

- __sized_by(size) — size denotes the size in bytes instead of number of elements

```
void *__sized_by(size) malloc(size_t size);
```

__counted_by variants for different use cases

```
void bzero(void *__sized_by(n) s, size_t n);
```

- __sized_by(size) — size denotes the size in bytes instead of number of elements

```
void *__sized_by_or_null(size) malloc(size_t size);
```

- __counted_by_or_null(), __sized_by_or_null() — allows NULL pointer with arbitrary count/size

Bounds are validated during initialization

- The compiler emits a bounds-check when a __counted_by pointer is initialized

```
void foo(int *__counted_by(len) p, int len);

void bar(int n) {
  int array[42];

  // bounds-check (n >= 0 && n <= 42)
  foo(array, n);
}
```

Single element pointers

- __single denotes a pointer that points to a single element
- Can be null
- Normal C pointer but with compile-time restrictions

```
void foo(int *__single p, int n) {
  p[42]; // compile error (p only has one element)
  p++;   // compile error (p would be invalid after p++)
  p[n];  // compile error (dynamic index likely a mistake)
  p[0];  // ok
  *p;    // ok
}
```

__bidi_indexable tracks both upper and lower bounds

- __bidi_indexable transforms a plain pointer into a wide pointer

```
struct wide_ptr {
  char *ptr;
  char *upper_bound;
  char *lower_bound;
};

void foo(char *__bidi_indexable p);

void foo(struct wide_ptr p);
```

__bidi_indexable can be modified without checks

- The compiler emits a bounds-check when the pointer is dereferenced

```
void foo(char *__bidi_indexable p) {

  p += 42;      // ok (p.ptr += 42, bounds unchanged)
  p -= 100;     // ok (p.ptr -= 100, bounds unchanged)
  char x = *p; // bounds-check (p.ptr >= p.lower_bound &&
                                 p.ptr < p.upper_bound)
}
```

__bidi_indexable is easy to use but breaks ABI

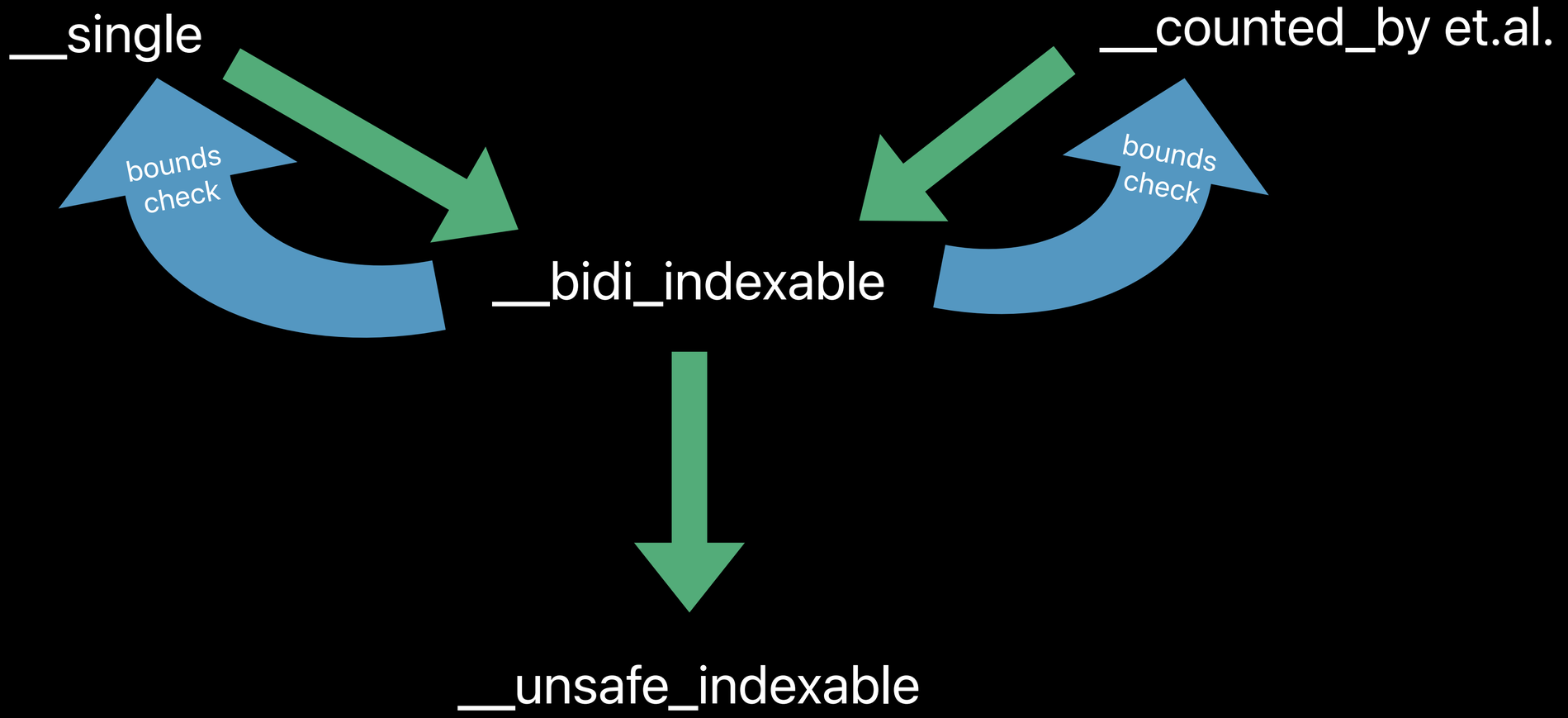
```
struct wide_ptr {
  char *ptr;
  char *upper_bound;
  char *lower_bound;
};
```

- Few compiler restrictions
- Wide pointer takes 3x the size of a raw pointer
- Not ABI compatible

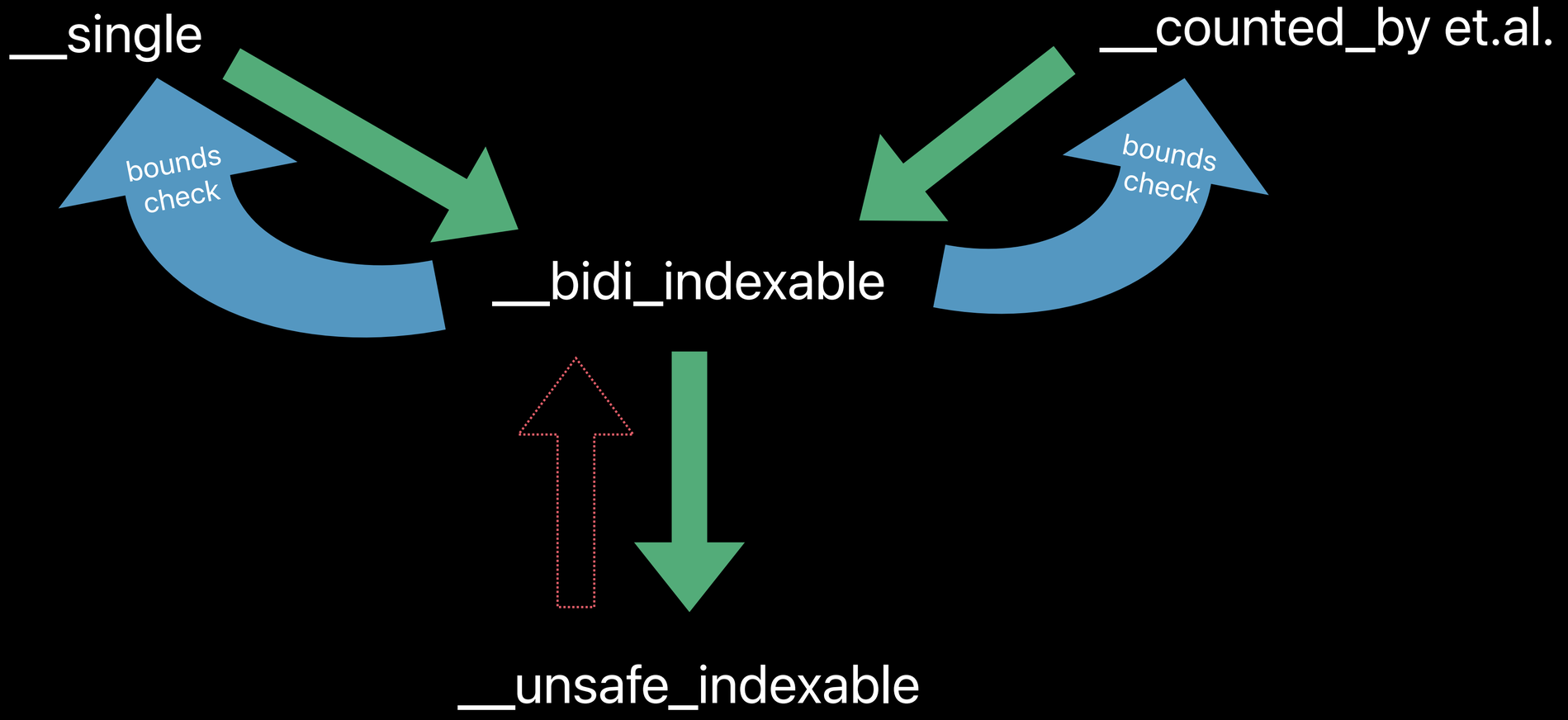
__unsafe_indexable

- Unsafe escape hatch
- Just like regular C pointers
  - Pointer arithmetic is allowed
  - No checks

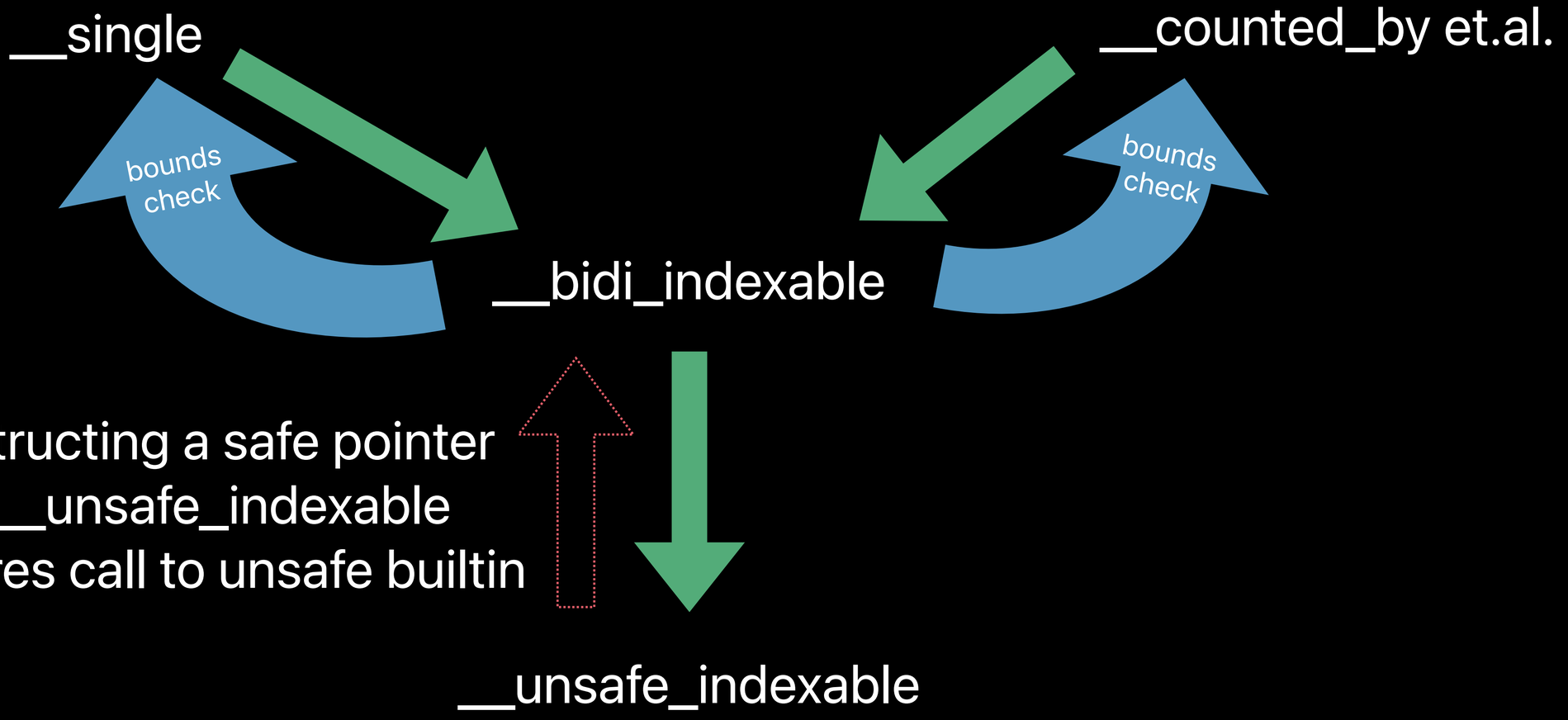
Converting between pointer kinds maintains invariants



Converting between pointer kinds maintains invariants



Converting between pointer kinds maintains invariants



Default annotations

- The defaults are secure by default and preserve ABI
- Sensible defaults reduce adoption time
- ABI visible pointers are __single by default
  - except when declared in system headers, then they are __unsafe_indexable
  - include 3rd party libraries as system headers to prevent errors
- Local pointers are __bidi_indexable by default

```
void foo(int *p1) { // p1 is __single
  int *p2 = ...;    // p2 is __bidi_indexable
}
```

Additional resources

- EuroLLVM 2023 keynote on -fbounds-safety
- https://clang.llvm.org/docs/BoundsSafety.html

Live adoption

Incremental adoption process

Incremental adoption process

1. Enable -fbounds-safety for a single C file

Incremental adoption process

1. Enable -fbounds-safety for a single C file
   a) Fix compilation errors

Incremental adoption process

1. Enable -fbounds-safety for a single C file
   a) Fix compilation errors
   b) Fix test failures - good test coverage is essential

Incremental adoption process

1. Enable -fbounds-safety for a single C file
   a) Fix compilation errors
   b) Fix test failures - good test coverage is essential
2. Repeat 1. until -fbounds-safety is enabled everywhere

Incremental adoption process

1. Enable -fbounds-safety for a single C file
   a) Fix compilation errors
   b) Fix test failures - good test coverage is essential
2. Repeat 1. until -fbounds-safety is enabled everywhere
3. Benchmark performance, measure binary size

Incremental adoption process

1. Enable -fbounds-safety for a single C file
   a) Fix compilation errors
   b) Fix test failures - good test coverage is essential
2. Repeat 1. until -fbounds-safety is enabled everywhere
3. Benchmark performance, measure binary size
   a) Optimize if needed - opt remarks can assist here

Libraries need to signal adoption

Libraries need to signal adoption

- System headers default to __unsafe_indexable for ABI visible pointers

Libraries need to signal adoption

- System headers default to __unsafe_indexable for ABI visible pointers
- Non-system headers default to __single for ABI visible pointers

Libraries need to signal adoption

- System headers default to __unsafe_indexable for ABI visible pointers
- Non-system headers default to __single for ABI visible pointers
- Mismatch when public headers of your library are included as system headers

Libraries need to signal adoption

- System headers default to __unsafe_indexable for ABI visible pointers
- Non-system headers default to __single for ABI visible pointers
- Mismatch when public headers of your library are included as system headers
- __ptrcheck_abi_assume_single() changes the default attribute to __single for the whole file

Libraries need to signal adoption

- System headers default to __unsafe_indexable for ABI visible pointers
- Non-system headers default to __single for ABI visible pointers
- Mismatch when public headers of your library are included as system headers
- __ptrcheck_abi_assume_single() changes the default attribute to __single for the whole file
- Public headers of your library should use __ptrcheck_abi_assume_single() to avoid the mismatch and signal that they adopted -fbounds-safety

Alternative: header-only adoption

Alternative: header-only adoption

- Lightweight alternative for libraries

Alternative: header-only adoption

- Lightweight alternative for libraries
- Only public interfaces are annotated

Alternative: header-only adoption

- Lightweight alternative for libraries
- Only public interfaces are annotated
  - Library implementation remains unsafe

Alternative: header-only adoption

- Lightweight alternative for libraries
- Only public interfaces are annotated
  - Library implementation remains unsafe
  - Clients adopting -fbounds-safety will get safe interface

Alternative: header-only adoption

- Lightweight alternative for libraries
- Only public interfaces are annotated
  - Library implementation remains unsafe
  - Clients adopting -fbounds-safety will get safe interface
  - Other clients pay no cost

Alternative: header-only adoption

- Lightweight alternative for libraries
- Only public interfaces are annotated
  - Library implementation remains unsafe
  - Clients adopting -fbounds-safety will get safe interface
  - Other clients pay no cost
- Example use case: C standard library

```
void *memcpy(void *__sized_by(n) dst, const void *__sized_by(n) src, size_t n);
```

Alternative: header-only adoption

- Lightweight alternative for libraries
- Only public interfaces are annotated
  - Library implementation remains unsafe
  - Clients adopting -fbounds-safety will get safe interface
  - Other clients pay no cost
- Example use case: C standard library

```
void *memcpy(void *__sized_by(n) dst, const void *__sized_by(n) src, size_t n);
```

- Also useful for safer interop from other languages

Alternative: header-only adoption

- Lightweight alternative for libraries
- Only public interfaces are annotated
  - Library implementation remains unsafe
  - Clients adopting -fbounds-safety will get safe interface
  - Other clients pay no cost
- Example use case: C standard library

```
void *memcpy(void *__sized_by(n) dst, const void *__sized_by(n) src, size_t n);
```

- Also useful for safer interop from other languages
- Remember to add test case including each header with -fbounds-safety enabled

Demo

Demo

1. Enable -fbounds-safety for a single C file
   a) Fix compilation errors
   b) Fix test failures - good test coverage is essential
2. Repeat 1. until -fbounds-safety is enabled everywhere
3. Benchmark performance, measure binary size
   a) Optimize if needed - opt remarks can assist here

Examples of adoption

Examples of adoption

- https://github.com/apple/sample-fbounds-safety-adoption
  - GIFLIB

Examples of adoption

- https://github.com/apple/sample-fbounds-safety-adoption
  - GIFLIB
- https://github.com/apple-oss-distributions/xnu

Summary

- While safe languages are great, securing existing code bases in unsafe languages is also necessary
- Incremental adoption and low adoption cost make this tractable even for large code bases
- Check out Devin Coughlin's keynote tomorrow: “A Recipe for Eliminating Entire Classes of Memory Safety Vulnerabilities in C and C++ ”
- Try it out and give us feedback!
  - Toolchain available at https://www.swift.org/install
  - #fbounds-safety on LLVM Discord