# Memory Safety in Systems Languages

## Major Area Exam

Michael Christensen

June 11, 2018

**Committee:**

Ben Hardekopf (🪑)    Tim Sherwood    Rich Wolski

# Outline

# Motivation

# What is a System?

Infrastructure software upon which applications are built

# What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

# What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

Embedded Systems,

# What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

Embedded Systems, Compilers,

# What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

Embedded Systems, Compilers, Garbage Collectors,

# What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

Embedded Systems, Compilers, Garbage Collectors, Device Drivers,

# What is a System?

Infrastructure software upon which applications are built

Operating Systems

- Process abstraction
- Multiplex physical hardware resources
- Partition and abstract **memory**

Embedded Systems, Compilers, Garbage Collectors, Device Drivers, File Systems

# Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...

# Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard

## Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control

# Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control

# Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers

## Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant

# Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy

# Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard

## Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard
  - Unchecked array operations

## Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard
  - Unchecked array operations
  - Pointers $\equiv$ arrays

# Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard
  - Unchecked array operations
  - Pointers $\equiv$ arrays
  - Unsafe casts

# Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard
  - Unchecked array operations
  - Pointers $\equiv$ arrays
  - Unsafe casts
  - Aliasing

# Systems Languages

- Past systems languages: ALGOL, PL/I, Fortran, BCPL/B, C, Mesa/Cedar, Pascal/Modula-2/Oberon, C++, ...
- C: the de-facto standard
  - Data structure representation control
  - Memory management control
  - Complete mutability via pointers
  - Performant
  - Legacy
- C: the unsafe standard
  - Unchecked array operations
  - Pointers $\equiv$ arrays
  - Unsafe casts
  - Aliasing
  - Undefined behavior

# Memory Safety

- Memory safety error: reads or writes outside the referent's storage

# Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds

# Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime

# Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is

# Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is
  - Efficient and expressive

# Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is
  - Efficient and expressive
  - Purely static

# Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is
  - Efficient and expressive
  - Purely static
  - Precise

# Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is
  - Efficient and expressive
  - Purely static
  - Precise
  - Automatic

# Memory Safety

- Memory safety error: reads or writes outside the referent's storage
  - Spatial: outside referent's address bounds
  - Temporal: outside referent's lifetime
- Ideal technique is
  - Efficient and expressive
  - Purely static
  - Precise
  - Automatic
- Memory errors become type errors, management happens at compile-time

# Outline

# Spatial Safety

Always access within object's bounds

# Spatial Safety

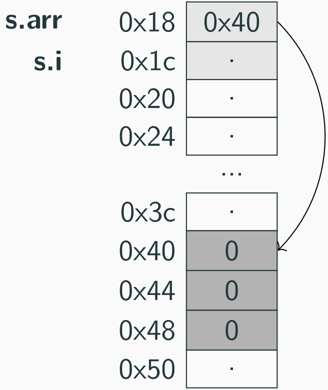
Always access within object's bounds

```
struct { int *arr; int i; } s;
s.arr = calloc(3, sizeof(int));
```

# Spatial Safety

Always access within object's bounds

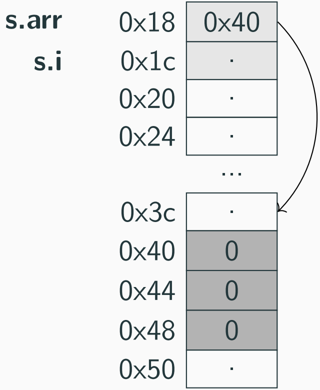
```
struct { int *arr; int i; } s;
s.arr = calloc(3, sizeof(int));
```



# Spatial Safety

Always access within object's bounds

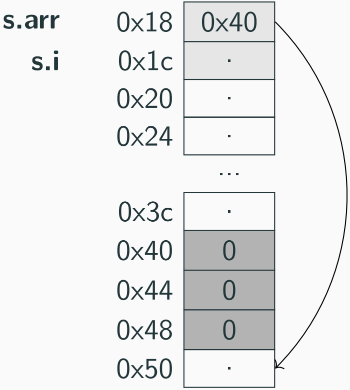
```
struct { int *arr; int i; } s;
s.arr = calloc(3, sizeof(int));
```



# Spatial Safety

Always access within object's bounds

```
struct { int *arr; int i; } s;
s.arr = calloc(3, sizeof(int));
```



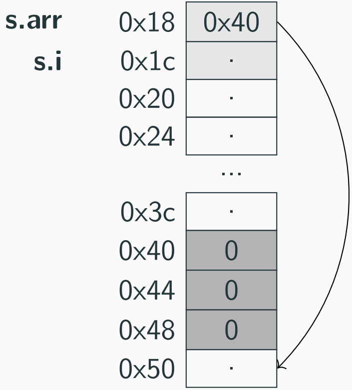
## Spatial Safety

Always access within object's bounds

Some approaches:

- Fat Pointers and Shadow Structures
- Referent Objects
- Dependent Types

```
struct { int *arr; int i; } s;
s.arr = calloc(3, sizeof(int));
```



# Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

# Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

# Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

# Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

## Spatial Safety Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

# Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

# Spatial Safety Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

# Spatial Safety Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

Potential pointer dereference problems:

- Null
- Uninitialized
- Out-of-bounds
- Manufactured

# Outline

# Fat Pointers

# Fat Pointers

- Added base and bound addresses

# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions

# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences

# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:

# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata

## Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety

# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows

# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:

# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility

## Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation

# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead

# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

# Fat Pointers

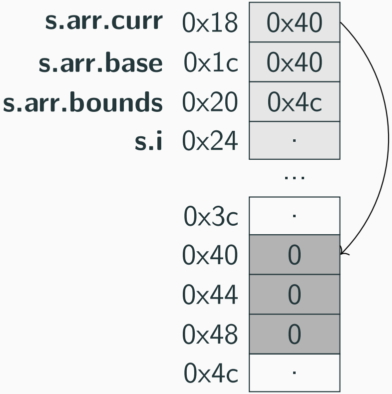
- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

```
struct fptr { int *curr;
     int *base; int *bound; };
struct { struct fptr arr; int i; } s;
s.arr.curr = calloc(3, sizeof(int));
```

# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

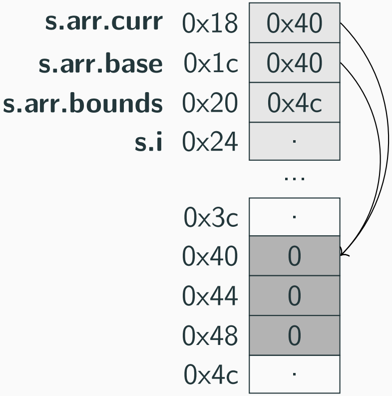
```
struct fptr { int *curr;
     int *base; int *bound; };
struct { struct fptr arr; int i; } s;
s.arr.curr = calloc(3, sizeof(int));
```



# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

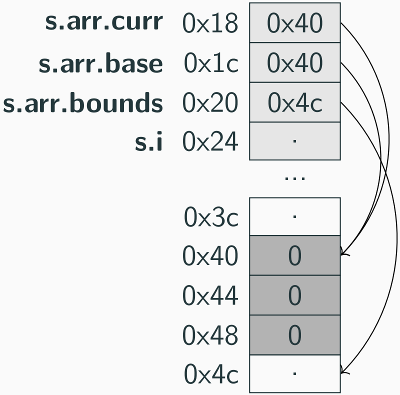
```
struct fptr { int *curr;
     int *base; int *bound; };
struct { struct fptr arr; int i; } s;
s.arr.curr = calloc(3, sizeof(int));
```



# Fat Pointers

- Added base and bound addresses
- Instrument all pointers and functions
- Insert runtime checks before dereferences
- Advantages:
  - Quickly find and retrieve metadata
  - Complete spatial safety
  - No sub-object overflows
- Disadvantages:
  - Breaks binary compatibility
  - Metadata propagation
  - Code bloat, memory usage, runtime overhead
  - Unsafe casts overwriting metadata

```
struct fptr { int *curr;
     int *base; int *bound; };
struct { struct fptr arr; int i; } s;
s.arr.curr = calloc(3, sizeof(int));
```



# Fat Pointer Approaches

SafeC[1]:

- Safe pointers have value, base, and size
- Complete spatial safety, if transparent storage management and no safe pointer attribute manipulation
- 275% space overhead, 2-6x runtime overhead, 0.35-3x code size overhead
- Some static optimization based on still-valid previous checks

[1] Austin, Breach, and Sohi, "Efficient Detection of All Pointer and Array Access Errors", 1994.

[2] Jim et al., "Cyclone: A Safe Dialect of C.", 2002.

# Fat Pointer Approaches

SafeC[1]:

- Safe pointers have value, base, and size
- Complete spatial safety, if transparent storage management and no safe pointer attribute manipulation
- 275% space overhead, 2-6x runtime overhead, 0.35-3x code size overhead
- Some static optimization based on still-valid previous checks

Cyclone[2]:

- Annotations for non-array vs array pointers (can specify size)
- Tagged unions and automatic tag injection

[1] Austin, Breach, and Sohi, "Efficient Detection of All Pointer and Array Access Errors", 1994.
[2] Jim et al., "Cyclone: A Safe Dialect of C.", 2002.

# Fat Pointer Approaches

CCured[3]

- Separate pointers on usage (SAFE, SEQ, WILD)
- Whole-program inference to find as many SAFE then SEQ pointers as possible
- Reduce WILD pointers[4] using physical subtyping[5] for upcasts
- Special pointer RTTI carrying runtime type for downcasts

---

[3] Necula, McPeak, and Weimer, "CCured", 2002.

[4] Necula, Condit, et al., "CCured", 2005.

[5] Siff et al., "Coping with Type Casts in C", 1999.

[6] Oiwa, "Implementation of the Memory-safe Full ANSI-C Compiler", 2009.

# Fat Pointer Approaches

CCured[3]

- Separate pointers on usage (SAFE, SEQ, WILD)
- Whole-program inference to find as many SAFE then SEQ pointers as possible
- Reduce WILD pointers[4] using physical subtyping[5] for upcasts
- Special pointer RTTI carrying runtime type for downcasts

Fail-Safe C[6]:

- Combines fat pointers w/ fat integers and virtual structure offsets

---

[3] Necula, McPeak, and Weimer, "CCured", 2002.

[4] Necula, Condit, et al., "CCured", 2005.

[5] Siff et al., "Coping with Type Casts in C", 1999.

[6] Oiwa, "Implementation of the Memory-safe Full ANSI-C Compiler", 2009.

## Fat Pointers Preventing Spatial Errors

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {



        if (*p == token) break;


        p++;
    }

    return (*p == token);
}
```

# Fat Pointers Preventing Spatial Errors

```c
1  int find_token(int *data,
2                 int *end,
3                 int token)
4  {
5      int *p = data;
6      while (p < end) {
7
8
9
10         if (*p == token) break;
11
12
13         p++;
14     }
15
16     return (*p == token);
17 }
```

```c
1  int find_token(int *SEQ data,
2                 int *SAFE end,
3                 int token)
4  {
5      int *SEQ p = data;
6      while (p.cur < end) {
7          assert(p.base != 0 &&
8                 0 <= p.cur &&
9                 p.cur < p.bound);
10         if (*p.cur == token) break;
11         p.cur = p.cur + (1 * sizeof(int));
12         p.base  = p.base;   // optimized out
13         p.bound = p.bound;  // "            "
14     }
15     ...(repeat lines 7-9)...
16     return (*p.cur == token);
17 }
```

# Fat Pointers Preventing Spatial Errors

```c
1   int find_token(int *data,
2                  int *end,
3                  int token)
4   {
5       int *p = data;
6       while (p < end) {
7
8
9
10          if (*p == token) break;
11
12
13          p++;
14      }
15
16      return (*p == token);
17  }
```

```c
1   int find_token(int *SEQ data,
2                  int *SAFE end,
3                  int token)
4   {
5       int *SEQ p = data;
6       while (p.cur < end) {
7           assert(p.base != 0 &&
8                  0 <= p.cur &&
9                  p.cur < p.bound);
10          if (*p.cur == token) break;
11          p.cur = p.cur + (1 * sizeof(int));
12          p.base  = p.base;   // optimized out
13          p.bound = p.bound;  // "            "
14      }
15      ...(repeat lines 7-9)...
16      return (*p.cur == token);
17  }
```

# Fat Pointers Preventing Spatial Errors

```c
1   int find_token(int *data,
2                  int *end,
3                  int token)
4   {
5       int *p = data;
6       while (p < end) {
7
8
9
10          if (*p == token) break;
11
12
13          p++;
14      }
15
16      return (*p == token);
17  }
```

```c
1   int find_token(int *SEQ data,
2                  int *SAFE end,
3                  int token)
4   {
5       int *SEQ p = data;
6       while (p.cur < end) {
7           assert(p.base != 0 &&
8                  0 <= p.cur &&
9                  p.cur < p.bound);
10          if (*p.cur == token) break;
11          p.cur = p.cur + (1 * sizeof(int));
12          p.base  = p.base;   // optimized out
13          p.bound = p.bound;  // "            "
14      }
15      ...(repeat lines 7-9)...
16      return (*p.cur == token);
17  }
```

# Fat Pointers Preventing Spatial Errors

```
1   int find_token(int *data,
2                  int *end,
3                  int token)
4   {
5       int *p = data;
6       while (p < end) {
7
8
9
10          if (*p == token) break;
11
12
13          p++;
14      }
15
16      return (*p == token);
17  }
```

```
1   int find_token(int *SEQ data,
2                  int *SAFE end,
3                  int token)
4   {
5       int *SEQ p = data;
6       while (p.cur < end) {
7           assert(p.base != 0 &&
8                  0 <= p.cur &&
9                  p.cur < p.bound);
10          if (*p.cur == token) break;
11          p.cur = p.cur + (1 * sizeof(int));
12          p.base  = p.base;   // optimized out
13          p.bound = p.bound;  // "            "
14      }
15      ...(repeat lines 7-9)...
16      return (*p.cur == token);
17  }
```

# Fat Pointers Preventing Spatial Errors

```c
1  int find_token(int *data,
2                 int *end,
3                 int token)
4  {
5      int *p = data;
6      while (p < end) {
7
8
9
10         if (*p == token) break;
11
12
13         p++;
14     }
15
16     return (*p == token);
17 }
```

```c
1  int find_token(int *SEQ data,
2                 int *SAFE end,
3                 int token)
4  {
5      int *SEQ p = data;
6      while (p.cur < end) {
7          assert(p.base != 0 &&
8                 0 <= p.cur &&
9                 p.cur < p.bound);
10         if (*p.cur == token) break;
11         p.cur = p.cur + (1 * sizeof(int));
12         p.base  = p.base;   // optimized out
13         p.bound = p.bound;  // "            "
14     }
15     ...(repeat lines 7-9)...
16     return (*p.cur == token);
17 }
```

# Fat Pointers Preventing Spatial Errors

```
1   int find_token(int *data,
2                  int *end,
3                  int token)
4   {
5       int *p = data;
6       while (p < end) {
7
8
9
10          if (*p == token) break;
11
12
13          p++;
14      }
15
16      return (*p == token);
17  }
```

```
1   int find_token(int *SEQ data,
2                  int *SAFE end,
3                  int token)
4   {
5       int *SEQ p = data;
6       while (p.cur < end) {
7           assert(p.base != 0 &&
8                  0 <= p.cur &&
9                  p.cur < p.bound);
10          if (*p.cur == token) break;
11          p.cur = p.cur + (1 * sizeof(int));
12          p.base  = p.base;   // optimized out
13          p.bound = p.bound;  // "           "
14      }
15      ...(repeat lines 7-9)...
16      return (*p.cur == token);
17  }
```

# Pointer-Based – Shadow Structures

MSCC[7]

[7] Xu, DuVarney, and Sekar, "An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs", 2004.

# Pointer-Based – Shadow Structures

MSCC[7]

- *Split* metadata from pointer, preserving layout
- Every value has linked shadow structure mirroring entire data structure
- Transform every function call to take additional metadata parameters
- Wrappers for external functions; cannot detect memory errors

[7] Xu, DuVarney, and Sekar, "An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs", 2004.

# Shadow Structures Example

```
int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;

    while (p < end) {

        if (*p == token) break;
        p++;
    }

    return (*p == token);
}
```

# Shadow Structures Example

```c
int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
struct ptr_info {
    void *base;
    unsigned long bound;
};
int find_token(
  int *data, ptr_info *data_info,
  int *end,  ptr_info *end_info,
  int token)
{
    int *p = data;
    ptr_info  p_info = *data_info;
    while (p < end) {
        CHECK_SPATIAL(p, sizeof(*p), p_info);
        if (*p == token) break;
        p++;
    }
    CHECK_SPATIAL(p, sizeof(*p), p_info);
    return (*p == token);
}
```

# Shadow Structures Example

```c
int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;

    while (p < end) {

        if (*p == token) break;
        p++;
    }

    return (*p == token);
}
```

```c
struct ptr_info {
    void *base;
    unsigned long bound;
};
int find_token(
  int *data, ptr_info *data_info,
  int *end,  ptr_info *end_info,
  int token)
{
    int *p = data;
    ptr_info  p_info = *data_info;
    while (p < end) {
        CHECK_SPATIAL(p, sizeof(*p), p_info);
        if (*p == token) break;
        p++;
    }
    CHECK_SPATIAL(p, sizeof(*p), p_info);
    return (*p == token);
}
```

# Shadow Structures Example

```c
int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;

    while (p < end) {

        if (*p == token) break;
        p++;
    }

    return (*p == token);
}
```

```c
struct ptr_info {
    void *base;
    unsigned long bound;
};
int find_token(
  int *data, ptr_info *data_info,
  int *end,  ptr_info *end_info,
  int token)
{
    int *p = data;
    ptr_info  p_info = *data_info;
    while (p < end) {
        CHECK_SPATIAL(p, sizeof(*p), p_info);
        if (*p == token) break;
        p++;
    }
    CHECK_SPATIAL(p, sizeof(*p), p_info);
    return (*p == token);
}
```

# Shadow Structures Example

```c
int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
struct ptr_info {
    void *base;
    unsigned long bound;
};
int find_token(
  int *data, ptr_info *data_info,
  int *end,  ptr_info *end_info,
  int token)
{
    int *p = data;
    ptr_info  p_info = *data_info;
    while (p < end) {
        CHECK_SPATIAL(p, sizeof(*p), p_info);
        if (*p == token) break;
        p++;
    }
    CHECK_SPATIAL(p, sizeof(*p), p_info);
    return (*p == token);
}
```

# Shadow Structures Example

```c
int find_token(
  int *data,
  int *end,
  int token)
{
    int *p = data;

    while (p < end) {

        if (*p == token) break;
        p++;
    }

    return (*p == token);
}
```

```c
struct ptr_info {
    void *base;
    unsigned long bound;
};
int find_token(
  int *data, ptr_info *data_info,
  int *end,  ptr_info *end_info,
  int token)
{
    int *p = data;
    ptr_info  p_info = *data_info;
    while (p < end) {
        CHECK_SPATIAL(p, sizeof(*p), p_info);
        if (*p == token) break;
        p++;
    }
    CHECK_SPATIAL(p, sizeof(*p), p_info);
    return (*p == token);
}
```

# Outline

# Referent Objects

## Objects[8,9]

- Metadata about objects, not pointers

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

# Referent Objects

## Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

# Referent Objects

## Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata

[8] Jones and Kelly, “Backwards-compatible bounds checking for arrays and pointers in C programs”, 1997.

[9] Ruwase and Lam, “A Practical Dynamic Buffer Overflow Detector”, 2004.

# Referent Objects

## Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata
- Bounds check on pointer arithmetic

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

# Referent Objects

## Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata
- Bounds check on pointer arithmetic
- 2-12x overhead

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

# Referent Objects

## Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata
- Bounds check on pointer arithmetic
- 2-12x overhead

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

# Referent Objects

Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata
- Bounds check on pointer arithmetic
- 2-12x overhead

Advantages:

- Compatible with uninstrumented code

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

# Referent Objects

## Objects[8,9]

- Metadata about objects, not pointers
- Global database relates every allocated address to corresponding object metadata
- Every pointer to same object shares same metadata
- Bounds check on pointer arithmetic
- 2-12x overhead

Advantages:

- Compatible with uninstrumented code

Disadvantages:

- Special mechanisms to handle legal OOB pointers
- Splay-tree object lookup overhead
- Incomplete spatial safety: sub-object overflows

[8] Jones and Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs", 1997.

[9] Ruwase and Lam, "A Practical Dynamic Buffer Overflow Detector", 2004.

# The Sub-Object Problem

```
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

# The Sub-Object Problem

```
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

# The Sub-Object Problem

```c
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

# The Sub-Object Problem

```c
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

## The Sub-Object Problem

```
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

# The Sub-Object Problem

```c
struct node {char str[3]; void (*func)(); };
struct node *n = (struct node *) malloc(sizeof(node));
char *s = n.str;
strcpy(s, "bad!");
```

- `n` and `s` have the same address ⇒ map to same object in global database
- `strcpy` will see `s`'s size as that of `n`

# Referent Objects Approaches

SafeCode[10]

- Use automatic pool allocation (APA)[11]
- Use separate, smaller data structures to store bounds metadata for each partition
- 1.2x overhead

[10] Dhurjati and Adve, "Backwards-compatible Array Bounds Checking for C with Very Low Overhead", 2006.

[11] Lattner and Adve, "Automatic Pool Allocation", 2005.

[12] Akritidis et al., "Baggy Bounds Checking", 2009.

# Referent Objects Approaches

SafeCode[10]

- Use automatic pool allocation (APA)[11]
- Use separate, smaller data structures to store bounds metadata for each partition
- 1.2x overhead

Baggy Bounds Checking (BBC)[12]

- Compact bounds representation and efficient way to look up object bounds
- Align base addresses to be multiple of padded size
- Replace splay tree with small lookup table
- 0.6x overhead

---

[10] Dhurjati and Adve, "Backwards-compatible Array Bounds Checking for C with Very Low Overhead", 2006.

[11] Lattner and Adve, "Automatic Pool Allocation", 2005.

[12] Akritidis et al., "Baggy Bounds Checking", 2009.

# Referent Objects Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

# Referent Objects Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        int *q = p + 1;
        int size = 1 << TABLE[p>>4];
        int base = p & ~(size - 1);
        assert(q >= base && q - base < size);
        p++;
    }
    return (*p == token);
}
```

# Referent Objects Example

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        int *q = p + 1;
        int size = 1 << TABLE[p>>4];
        int base = p & ~(size - 1);
        assert(q >= base && q - base < size);
        p++;
    }
    return (*p == token);
}
```

# Referent Objects Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;



        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        int *q = p + 1;
        int size = 1 << TABLE[p>>4];
        int base = p & ~(size - 1);
        assert(q >= base && q - base < size);
        p++;
    }
    return (*p == token);
}
```

# Referent Objects Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        int *q = p + 1;
        int size = 1 << TABLE[p>>4];
        int base = p & ~(size - 1);
        assert(q >= base && q - base < size);
        p++;
    }
    return (*p == token);
}
```

# Referent Objects Example

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        int *q = p + 1;
        int size = 1 << TABLE[p>>4];
        int base = p & ~(size - 1);
        assert(q >= base && q - base < size);
        p++;
    }
    return (*p == token);
}
```

# Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table

[13] Nagarakatte, Zhao, Milo MK Martin, et al., “SoftBound”, 2009.

# Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches

[13] Nagarakatte, Zhao, Milo MK Martin, et al., “SoftBound”, 2009.

# Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches
- Source compatibility, separate compilation of object-based approaches

[13] Nagarakatte, Zhao, Milo MK Martin, et al., "SoftBound", 2009.

# Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches
- Source compatibility, separate compilation of object-based approaches
- Runtime bounds checks on each dereference

[13] Nagarakatte, Zhao, Milo MK Martin, et al., "SoftBound", 2009.

# Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches
- Source compatibility, separate compilation of object-based approaches
- Runtime bounds checks on each dereference
- Propagate metadata as extra arguments

[13] Nagarakatte, Zhao, Milo MK Martin, et al., "SoftBound", 2009.

# Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches
- Source compatibility, separate compilation of object-based approaches
- Runtime bounds checks on each dereference
- Propagate metadata as extra arguments
- Arbitrary casts allowed

[13] Nagarakatte, Zhao, Milo MK Martin, et al., "SoftBound", 2009.

# Softbound

Softbound[13]

- Base and bound metadata for each pointer, stored in disjoint metadata table
- Total spatial safety of pointer-based approaches
- Source compatibility, separate compilation of object-based approaches
- Runtime bounds checks on each dereference
- Propagate metadata as extra arguments
- Arbitrary casts allowed
- 67% overhead

[13] Nagarakatte, Zhao, Milo MK Martin, et al., “SoftBound”, 2009.

# Outline

# Dependent Types

Dependent types are *typed-valued functions*[14]

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

# Dependent Types

Dependent types are *typed-valued functions*[14]

$$\texttt{Vector : Nat} \rightarrow \texttt{Type} \rightarrow \texttt{Type}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

# Dependent Types

Dependent types are *typed-valued functions*[14]

$$\texttt{Vector} : \texttt{Nat} \rightarrow \texttt{Type} \rightarrow \texttt{Type}$$
$$\texttt{nil} : \texttt{Vector 0 a}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

# Dependent Types

Dependent types are *typed-valued functions*[14]

$$\begin{aligned}\texttt{Vector} &: \texttt{Nat} \rightarrow \texttt{Type} \rightarrow \texttt{Type}\\ \texttt{nil} &: \texttt{Vector 0 a}\\ \texttt{cons} &: \Pi\texttt{n:Nat.a} \rightarrow \texttt{Vector n a} \rightarrow \texttt{Vector (n+1) a}\end{aligned}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

# Dependent Types

Dependent types are *typed-valued functions*[14]

$$\begin{aligned}\texttt{Vector} &: \texttt{Nat} \rightarrow \texttt{Type} \rightarrow \texttt{Type}\\ \texttt{nil} &: \texttt{Vector 0 a}\\ \texttt{cons} &: \Pi\texttt{n:Nat.a} \rightarrow \texttt{Vector n a} \rightarrow \texttt{Vector (n+1) a}\\ \texttt{(cons 'a' (cons 'b' nil))} &: \texttt{Vector 2 Char}\end{aligned}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

# Dependent Types

Dependent types are *typed-valued functions*[14]

$$\texttt{Vector} : \texttt{Nat} \to \texttt{Type} \to \texttt{Type}$$
$$\texttt{nil} : \texttt{Vector 0 a}$$
$$\texttt{cons} : \Pi\texttt{n:Nat.a} \to \texttt{Vector n a} \to \texttt{Vector (n+1) a}$$
$$\texttt{(cons 'a' (cons 'b' nil))} : \texttt{Vector 2 Char}$$
$$\texttt{head} : \Pi\texttt{n:Nat.Vector a (n+1) -> a}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

# Dependent Types

Dependent types are *typed-valued functions*[14]

$$\texttt{Vector} : \texttt{Nat} \rightarrow \texttt{Type} \rightarrow \texttt{Type}$$

$$\texttt{nil} : \texttt{Vector 0 a}$$

$$\texttt{cons} : \Pi\texttt{n:Nat.a} \rightarrow \texttt{Vector n a} \rightarrow \texttt{Vector (n+1) a}$$

$$\texttt{(cons 'a' (cons 'b' nil))} : \texttt{Vector 2 Char}$$

$$\texttt{head} : \Pi\texttt{n:Nat.Vector a (n+1) -> a}$$

$$\texttt{head nil}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

# Dependent Types

Dependent types are *typed-valued functions*[14]

$$\begin{aligned}
\texttt{Vector} &: \texttt{Nat} \to \texttt{Type} \to \texttt{Type} \\
\texttt{nil} &: \texttt{Vector 0 a} \\
\texttt{cons} &: \Pi \texttt{n:Nat.a} \to \texttt{Vector n a} \to \texttt{Vector (n+1) a} \\
\texttt{(cons 'a' (cons 'b' nil))} &: \texttt{Vector 2 Char} \\
\texttt{head} &: \Pi \texttt{n:Nat.Vector a (n+1) -> a} \\
\texttt{head nil} &\Rightarrow \text{Rejected!}
\end{aligned}$$

[14] Pierce, *Advanced topics in types and programming languages*, 2005.

[15] Martin-Löf, "Constructive mathematics and computer programming", 1984.

# Dependent Types

Dependent types are *typed-valued functions*[14]

$$\begin{aligned}
\texttt{Vector} &: \texttt{Nat} \rightarrow \texttt{Type} \rightarrow \texttt{Type} \\
\texttt{nil} &: \texttt{Vector 0 a} \\
\texttt{cons} &: \Pi\texttt{n:Nat.a} \rightarrow \texttt{Vector n a} \rightarrow \texttt{Vector (n+1) a} \\
\texttt{(cons 'a' (cons 'b' nil))} &: \texttt{Vector 2 Char} \\
\texttt{head} &: \Pi\texttt{n:Nat.Vector a (n+1) -> a} \\
\texttt{head nil} &\Rightarrow \text{Rejected!}
\end{aligned}$$

- Based on type theory work by Martin-Löf[15]
- Undecidability of type checking: arbitrary computation to check type equality
- Work on defining equality and restricting forms of index terms

[14]Pierce, *Advanced topics in types and programming languages*, 2005.

[15]Martin-Löf, "Constructive mathematics and computer programming", 1984.

# Early Uses of Dependent Types

Dependent ML[16] and Cayenne[17]

- Reduce static array bound checking to constraint satisfiability
- DML uses *indexed types*: limit indices to linear integer and boolean expressions; compile-time decidable
- Cayenne has *no restrictions* on types: undecidability of arbitrary expression equivalence and thus type checking

Xanadu[18]

- Imperative environment
- Restrict index expressions in types to integer constraint domain

[16] Xi and Pfenning, "Eliminating Array Bound Checking Through Dependent Types", 1998.
[17] Augustsson, "Cayenne—a Language with Dependent Types", 1998.
[18] Xi, "Imperative programming with dependent types", 2000.

# Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

# Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

# Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope

---

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

# Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope
- Three-phase pass over annotated C programs, emits C code

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

# Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope
- Three-phase pass over annotated C programs, emits C code
  1. Automatic addition of bounds annotations for pointer types

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

# Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope
- Three-phase pass over annotated C programs, emits C code
  1. Automatic addition of bounds annotations for pointer types
  2. Flow-insensitive type checking (insert run-time checks; helps decidability)

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

# Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope
- Three-phase pass over annotated C programs, emits C code
  1. Automatic addition of bounds annotations for pointer types
  2. Flow-insensitive type checking (insert run-time checks; helps decidability)
  3. Flow-sensitive check optimization

---

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

# Dependent Types in Imperative Languages

SafeDrive[19] and Deputy[20,21]

- User-added annotations relating pointers to bounds
  - `safe`, `sentinel`, `count(n)`, `bound(lo,hi)`
  - Use constants/variables/field names in immediately enclosing scope
- Three-phase pass over annotated C programs, emits C code
  1. Automatic addition of bounds annotations for pointer types
  2. Flow-insensitive type checking (insert run-time checks; helps decidability)
  3. Flow-sensitive check optimization
- More C Support with dependent union tags , Safe TinyOS[22]

[19] Zhou et al., "SafeDrive", 2006.

[20] Condit et al., "Dependent Types for Low-Level Programming", 2007.

[21] Anderson, "Static Analysis of C for Hybrid Type Checking", 2007.

[22] Cooprider et al., "Efficient Memory Safety for TinyOS", 2007.

## Deputy Example for Spatial Safety

```
int find_token(int *data,
               int *end,
               int token)
{


    int *p = data;
    while (p < end) {


        if (*p == token) break;

        p++;
    }


    return (*p == token);
}
```

# Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

# Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{


    int *p = data;
    while (p < end) {


        if (*p == token) break;

        p++;
    }


    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

# Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

# Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{


    int *p = data;
    while (p < end) {


        if (*p == token) break;

        p++;
    }


    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

# Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{


    int *p = data;
    while (p < end) {


        if (*p == token) break;

        p++;
    }


    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

# Deputy Example for Spatial Safety

```c
int find_token(int *data,
               int *end,
               int token)
{
    int *p = data;
    while (p < end) {
        if (*p == token) break;
        p++;
    }
    return (*p == token);
}
```

```c
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

# Deputy Example for Spatial Safety

```
int find_token(int *data,
               int *end,
               int token)
{


    int *p = data;
    while (p < end) {


        if (*p == token) break;

        p++;
    }


    return (*p == token);
}
```

```
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

# Deputy Example for Spatial Safety

```
int find_token(int *data,
               int *end,
               int token)
{


    int *p = data;
    while (p < end) {


        if (*p == token) break;

        p++;
    }


    return (*p == token);
}
```

```
int find_token(int * bound(data, end) data,
               int * sentinel end,
               int token)
{
    assert(data != NULL);
    assert(end != NULL);
    int *p bound(p, end) = data;
    while (p < end) {
        assert(p != NULL);
        assert(p < end);
        if (*p == token) break;
        assert(p <= p + 1 <= end);
        p++;
    }
    assert(p != NULL);
    assert(p < end);
    return (*p == token);
}
```

# Abstract Syntax, For Your Consideration

$$x, y \in \text{Variables} \quad \text{op} \in \text{Binary ops} \quad n \in \text{Integers} \quad \text{comp} \in \text{Comparison Ops}$$

$$\begin{array}{rl} \textit{Ctors} & C ::= \text{int} \mid \text{ref} \mid \text{array} \\ \textit{Types} & \tau ::= C \mid \tau_1\ \tau_2 \mid \tau\ e \\ \textit{L-exprs} & l ::= x \mid *\ e \end{array}$$

# Abstract Syntax, For Your Consideration

$x, y \in$ Variables $\quad$ op $\in$ Binary ops $\quad$ $n \in$ Integers $\quad$ comp $\in$ Comparison Ops

$$\begin{array}{rl} \textit{Ctors} & C ::= \mathsf{int} \mid \mathsf{ref} \mid \mathsf{array} \\ \textit{Types} & \tau ::= C \mid \tau_1\ \tau_2 \mid \tau\ e \\ \textit{L-exprs} & l\ ::= x \mid *\, e \end{array}$$

$$\begin{array}{rl} \textit{Exprs} & e ::= n \mid l \mid e_1 \text{ op } e_2 \\ \textit{Cmds} & c ::= l := e \mid \mathsf{assert}(\gamma) \mid c_1 ; c_2 \mid \ldots \\ \text{Preds} & \gamma ::= e_1 \text{ comp } e_2 \mid \mathsf{true} \mid \gamma_1 \wedge \gamma_2 \end{array}$$

# Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

# Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau} \text{ (LOCAL NAME)} \qquad \frac{}{\Gamma \vdash_L n : \mathit{int}} \text{ (LOCAL NUM)}$$

# Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\boxed{\Gamma(x) = \tau}}{\boxed{\Gamma}\ \boxed{\vdash_L x}\ \boxed{: \tau}}\ (\textsc{Local Name}) \qquad \frac{}{\Gamma \vdash_L n : \mathit{int}}\ (\textsc{Local Num})$$

Non-local Expressions: $\Gamma \vdash e : \tau \Rightarrow \gamma$

# Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau}\ (\textsc{local name}) \qquad \frac{}{\Gamma \vdash_L n : \mathit{int}}\ (\textsc{local num})$$

Non-local Expressions: $\Gamma \vdash e : \tau \Rightarrow \gamma$

$$\frac{\Gamma \vdash e : \mathrm{ref}\ \tau \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma}\ (\textsc{deref})$$

# Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau}\ (\textsc{Local Name}) \qquad \frac{}{\Gamma \vdash_L n : \mathit{int}}\ (\textsc{Local Num})$$

Non-local Expressions: $\Gamma \vdash e : \tau \Rightarrow \gamma$

$$\frac{\Gamma \vdash e : \text{ref } \tau \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma}\ (\textsc{Deref})$$

Commands: $\Gamma \vdash c \Rightarrow c'$

# Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau}\ (\textsc{Local Name}) \qquad \frac{}{\Gamma \vdash_L n : \mathit{int}}\ (\textsc{Local Num})$$

Non-local Expressions: $\Gamma \vdash e : \tau \Rightarrow \gamma$

$$\frac{\Gamma \vdash e : \mathrm{ref}\ \tau \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma}\ (\textsc{Deref})$$

Commands: $\Gamma \vdash c \Rightarrow c'$

$$\frac{x \in \mathrm{Dom}(\Gamma) \qquad \forall (y : \tau_y) \in \Gamma,\ \Gamma \vdash y[e/x] : \tau_y[e/x] \Rightarrow \gamma_y}{\Gamma \vdash x := e \Rightarrow \mathrm{assert}(\bigwedge_{y \in \mathrm{Dom}(\Gamma)} \gamma_y);\ x := e}\ (\textsc{Var Write})$$

# Typing Rules, For Your Consideration

Local Expressions: $\Gamma \vdash_L e : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_L x : \tau}\ (\textsc{local name}) \qquad \frac{}{\Gamma \vdash_L n : \mathit{int}}\ (\textsc{local num})$$

Non-local Expressions: $\Gamma \vdash e : \tau \Rightarrow \gamma$

$$\frac{\Gamma \vdash e : \mathrm{ref}\ \tau \Rightarrow \gamma}{\Gamma \vdash *e : \tau \Rightarrow \gamma}\ (\textsc{deref})$$

Commands: $\Gamma \vdash c \Rightarrow c'$

$$\frac{x \in \mathrm{Dom}(\Gamma) \qquad \forall (y : \tau_y) \in \Gamma,\ \Gamma \vdash y[e/x] : \tau_y[e/x] \Rightarrow \gamma_y}{\Gamma \vdash x := e \Rightarrow \mathrm{assert}(\bigwedge_{y \in \mathrm{Dom}(\Gamma)} \gamma_y);\ \ x := e}\ (\textsc{var write})$$

# The Interesting Rules

Dereferencing

$$\frac{\Gamma \vdash e : \text{array } \tau \;\; e_{len} \Rightarrow \gamma_e}{\Gamma \vdash *e ; \tau \Rightarrow \gamma_e \wedge (0 < e_{len})} \text{ (ARRAY DEREF)}$$

# The Interesting Rules

Dereferencing

$$\frac{\Gamma \vdash e : \text{array } \tau \;\; e_{len} \Rightarrow \gamma_e}{\Gamma \vdash *e; \tau \Rightarrow \gamma_e \wedge (0 < e_{len})} \;(\textsc{Array Deref})$$

Arithmetic

$$\frac{\boxed{\Gamma}\boxed{\vdash e}\boxed{: \text{array } \tau \;\; e_{len} \Rightarrow \gamma_e} \qquad \boxed{\Gamma}\boxed{\vdash e'}\boxed{: \text{int} \Rightarrow \gamma_{e'}}}{\boxed{\Gamma}\boxed{\vdash e + e'}\boxed{: \text{array } \tau \;\; (e_{len} - e') \Rightarrow \gamma_e \wedge \gamma'_e \wedge (0 \leqslant e' \leqslant e_{len})}} \;(\textsc{Array Arith})$$

## Dependent Types in Imperative Languages

Týr[23]

- Augments LLVM IR with dependent pointer types
- Uses programmer annotations insert run-time bounds checks
- LLVM optimizations remove always-true checks; error if always-false

[23] De Araújo, Moreira, and Machado, "Týr", 2016.

[24] Ruef et al., "Checked C for Safety, Gradually", 2017.

[25] Protzenko et al., "Verified Low-level Programming Embedded in F*", 2017.

# Dependent Types in Imperative Languages

Týr[23]

- Augments LLVM IR with dependent pointer types
- Uses programmer annotations insert run-time bounds checks
- LLVM optimizations remove always-true checks; error if always-false

Checked C[24]

- Extend C with two *checked pointer types*: `_Ptr<T>` and `_Array_ptr<T>`
- Associated bounds expressions indicating where bounds are stored
- Isolate (un)safe code with *checked code regions*

[23] De Araújo, Moreira, and Machado, "Týr", 2016.

[24] Ruef et al., "Checked C for Safety, Gradually", 2017.

[25] Protzenko et al., "Verified Low-level Programming Embedded in F*", 2017.

# Dependent Types in Imperative Languages

Týr[23]

- Augments LLVM IR with dependent pointer types
- Uses programmer annotations insert run-time bounds checks
- LLVM optimizations remove always-true checks; error if always-false

Checked C[24]

- Extend C with two *checked pointer types*: `_Ptr<T>` and `_Array_ptr<T>`
- Associated bounds expressions indicating where bounds are stored
- Isolate (un)safe code with *checked code regions*

Low*[25]

- DSL for verified, efficient low-level programming in F*
- Write F* syntax against library modelling lower-level view of C memory

[23] De Araújo, Moreira, and Machado, "Týr", 2016.

[24] Ruef et al., "Checked C for Safety, Gradually", 2017.

[25] Protzenko et al., "Verified Low-level Programming Embedded in F*", 2017.

# Quick Spatial Recap

Spatial Safety

- Arrays and pointers

# Quick Spatial Recap

Spatial Safety

- Arrays and pointers
- Fat pointers

# Quick Spatial Recap

Spatial Safety

- Arrays and pointers
- Fat pointers
- Referent objects

# Quick Spatial Recap

Spatial Safety

- Arrays and pointers
- Fat pointers
- Referent objects
- Dependent types

# Outline

# Temporal Safety

Prevent accessing object that has been previously deallocated

# Temporal Safety

Prevent accessing object that has been previously deallocated

- Capabilities and locks
- Effects and regions
- Linear types and ownership

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
                //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
                //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```c
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
               //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```c
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
               //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
               //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
            //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Potential pointer dereference problems:

- Double frees
- Dangling pointers

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
                //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Potential pointer dereference problems:

- Double frees
- Dangling pointers

A Real Bug

- Linux Kernel in `ipc/mqueue.c`
- July 2017
- `https://bugzilla.redhat.com/show_bug.cgi?id=1470659`

# Temporal Safety Example

```
int attach(struct sock *sk) {
    if (sk->bad) {
        free(sk); return 1;
    }
    return 0;
}
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    while (n->try) {
        sock = malloc_sock(n->info);
        if (attach(sock)){
                //sock = NULL;
            break;
        }
    }
    if (sock) free(sock);
}
```

Potential pointer dereference problems:

- Double frees
- Dangling pointers

A Real Bug

- Linux Kernel in `ipc/mqueue.c`
- July 2017
- `https://bugzilla.redhat.com/show_bug.cgi?id=1470659`

Goals:

- Good: Detecting use-after-free
- Better: Eliminating free entirely

# A Comment on Garbage Collection

Garbage collection

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

# A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

# A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

# A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...
    - Non-zero overhead

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

# A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...
  - Non-zero overhead
  - Drag

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

# A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...
  - Non-zero overhead
  - Drag
  - Loss of real-time guarantees/predictability

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

# A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...
  - Non-zero overhead
  - Drag
  - Loss of real-time guarantees/predictability
  - Reduced reference locality, increased page fault/cache miss rates

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

# A Comment on Garbage Collection

Garbage collection

- Relinquish control of object location and layout to runtime
- Complete temporal safety, but...
  - Non-zero overhead
  - Drag
  - Loss of real-time guarantees/predictability
  - Reduced reference locality, increased page fault/cache miss rates
- Some spatial approaches (e.g. Fail-Safe C, CCured) use Boehm-Demers-Weister[26]

[26] Boehm and Weiser, "Garbage Collection in an Uncooperative Environment", 1988.

# Outline

# Capabilities and Locks

[27]Nagarakatte, Zhao, Milo M.K. Martin, et al., "CETS", 2010.

[28]Nagarakatte, M. M. K. Martin, and Zdancewic, "Everything You Want to Know About Pointer-Based Checking", 2015.

[29]Simpson and Barua, "MemSafe", 2013.

# Capabilities and Locks

- SafeC, MSCC
  - Unique capability associated with each memory block
  - Stored in capability store, marked invalid on free
  - Check if pointer's capability copy is still valid on dereference

[27] Nagarakatte, Zhao, Milo M.K. Martin, et al., "CETS", 2010.

[28] Nagarakatte, M. M. K. Martin, and Zdancewic, "Everything You Want to Know About Pointer-Based Checking", 2015.

[29] Simpson and Barua, "MemSafe", 2013.

# Capabilities and Locks

- SafeC, MSCC
  - Unique capability associated with each memory block
  - Stored in capability store, marked invalid on free
  - Check if pointer's capability copy is still valid on dereference
- CETS[27,28]
  - Each allocation has unique (never reused) key and lock address
  - Freeing allocated object changes value at lock location, so key and lock value don't match

[27] Nagarakatte, Zhao, Milo M.K. Martin, et al., "CETS", 2010.

[28] Nagarakatte, M. M. K. Martin, and Zdancewic, "Everything You Want to Know About Pointer-Based Checking", 2015.

[29] Simpson and Barua, "MemSafe", 2013.

# Capabilities and Locks

- SafeC, MSCC
  - Unique capability associated with each memory block
  - Stored in capability store, marked invalid on free
  - Check if pointer's capability copy is still valid on dereference
- CETS[27,28]
  - Each allocation has unique (never reused) key and lock address
  - Freeing allocated object changes value at lock location, so key and lock value don't match
- Memsafe[29]
  - Set bounds of deallocated pointer to invalid value

[27] Nagarakatte, Zhao, Milo M.K. Martin, et al., "CETS", 2010.

[28] Nagarakatte, M. M. K. Martin, and Zdancewic, "Everything You Want to Know About Pointer-Based Checking", 2015.

[29] Simpson and Barua, "MemSafe", 2013.

# Lock Example

```
int attach(struct sock *sk,
        key_t sk_key,
        lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

# Lock Example

```
int attach(struct sock *sk,
      key_t sk_key,
      lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

# Lock Example

```
int attach(struct sock *sk,
      key_t sk_key,
      lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

# Lock Example

```
int attach(struct sock *sk,
      key_t sk_key,
      lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

# Lock Example

```c
int attach(struct sock *sk,
      key_t sk_key,
      lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```c
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

# Lock Example

```
int attach(struct sock *sk,
      key_t sk_key,
      lock_t *sk_lock_addr) {
    if (sk_key != *sk_lock_addr)
      abort();
    if (sk->bad) {
        if (Freeable_ptrs_map.lookup(sk_key) != sk)
          abort();
        free(sk);
        *sk_lock_addr = INVALID_KEY;
        deallocate_lock(sk_lock_addr);
        return 1;
    }
    return 0;
}
```

```
void mq_notify(sigevent *n) {
    struct sock_t *sock;
    key_t sock_key;
    lock_t *sock_lock_addr;
    while (n->try) {
        sock = malloc_sock(n->info);
        sock_key = Next_key++;
        sock_lock_addr = allocate_lock();
        *(sock_lock_addr) = sock_key;
        Freeable_ptrs_map.insert(sock_key, sock);
        if (attach(sock)){
            break;
        }
    }
    if (sock) {
        if (Freeable_ptrs_map.lookup(sock_key) != sock)
          abort();
        free(sock);
        *(sock_lock_addr) = INVALID_KEY;
        deallocate_lock(sock_lock_addr);
    }
}
```

# Outline

## Effects

Effect types describe the *effects* of the computation leading to a value[30]

- Opening a file
- Modifying an object

[30] Pierce, *Advanced topics in types and programming languages*, 2005.

# Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.

[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

# Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.

[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

# Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use
- Identify referentially transparent expressions via a side effect lattice

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.

[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

# Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use
- Identify referentially transparent expressions via a side effect lattice
- Effect masking: inference to delimit regions of memory and their lifetimes

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.

[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

# Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use
- Identify referentially transparent expressions via a side effect lattice
- Effect masking: inference to delimit regions of memory and their lifetimes

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.

[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

# Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use
- Identify referentially transparent expressions via a side effect lattice
- Effect masking: inference to delimit regions of memory and their lifetimes

$\{W\ R^?\ A^?\}$ $\boxed{\text{Procedure}}$

$\{R\ A^?\}$ $\boxed{\text{Observer}}$

$\{A\}$ $\boxed{\text{Function}}$

$\{\}$ $\boxed{\text{Pure}}$

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.

[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

# Effects

Fluent Languages,[31] MFX[32]

- Mix functional and imperative languages
- Every expression has a effect class, restricting which subroutines or sublanguage features it may use
- Identify referentially transparent expressions via a side effect lattice
- Effect masking: inference to delimit regions of memory and their lifetimes

$\{W\ R^?\ A^?\}$ Procedure

$\{R\ A^?\}$ Observer

$\{A\}$ Function

$\{\}$ Pure

E.g.

- `update` is Procedure
- `nth` is Observer
- `arrayCreate` is Function
- `length` is Pure

[31] David K. Gifford and John M. Lucassen, "Integrating functional and imperative programming", 1986.

[32] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems", 1988.

# Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations

[33] F. Nielson and H. R. Nielson, “Type and Effect Systems”, 1999.

# Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

[33] F. Nielson and H. R. Nielson, “Type and Effect Systems”, 1999.

# Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

# Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \texttt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \\
\text{Types} & \tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \rightarrow \tau_2
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

## Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x := e_1\ \mathtt{in}\ e_2 \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \rightarrow \tau_2
\end{array}
$$

[33] F. Nielson and H. R. Nielson, “Type and Effect Systems”, 1999.

# Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$\text{Exprs} \quad e ::= c \mid x \mid \texttt{fn}_\pi\ x\ \Rightarrow e \mid e_1\ e_2 \mid \texttt{new}_\pi\ x\ :=\ e_1\ \texttt{in}\ e_2 \mid !x$$

$$\text{Types} \quad \tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

# Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x := e_1\ \mathtt{in}\ e_2 \mid\ !x \mid x := e \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \to \tau_2
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

## Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x := e_1\ \mathtt{in}\ e_2 \mid\ !x \mid x := e \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \rightarrow \tau_2 \mid \tau\ \mathtt{ref}
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

# Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$\text{Exprs} \quad e ::= c \mid x \mid \mathtt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x := e_1\ \mathtt{in}\ e_2 \mid !x \mid x := e$$

$$\text{Types} \quad \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \xrightarrow{\phi} \tau_2 \mid \tau\ \mathtt{ref}\ \varrho$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

# Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x\ \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x\ :=\ e_1\ \mathtt{in}\ e_2 \mid !x \mid x := e \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \xrightarrow{\phi} \tau_2 \mid \tau\ \mathtt{ref}\ \varrho \\
\text{Effect} & \phi ::= \{!\pi\} \mid \{\pi :=\} \mid \{\mathtt{new}\ \pi\} \mid \phi_1 \cup \phi_2 \mid \emptyset
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

# Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \ \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x\ :=\ e_1\ \mathtt{in}\ e_2 \mid !x \mid x := e \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \xrightarrow{\phi} \tau_2 \mid \tau\ \mathtt{ref}\ \varrho \\
\text{Effect} & \phi ::= \{!\pi\} \mid \{\pi :=\} \mid \{\mathtt{new}\ \pi\} \mid \phi_1 \cup \phi_2 \mid \varnothing \\
\text{PrgPts} & \varrho ::= \{\pi\} \mid \varrho_1 \cup \varrho_2 \mid \varnothing
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x\ \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x\ :=\ e_1\ \mathtt{in}\ e_2 \mid !x \mid x := e \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \xrightarrow{\phi} \tau_2 \mid \tau\ \mathtt{ref}\ \varrho \\
\text{Region} & \phi ::= \{!\pi\} \mid \{\pi :=\} \mid \{\mathtt{new}\ \pi\} \mid \phi_1 \cup \phi_2 \mid \emptyset \\
\text{PrgPts} & \varrho ::= \{\pi\} \mid \varrho_1 \cup \varrho_2 \mid \emptyset
\end{array}
$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

# Effects

Type and Effect Systems[33]

- Extend the simply-typed lambda calculus with annotations
- Various forms of analyses, incl. Side Effect Analysis and Region Inference

$$
\begin{array}{rl}
\text{Exprs} & e ::= c \mid x \mid \mathtt{fn}_\pi\ x \Rightarrow e \mid e_1\ e_2 \mid \mathtt{new}_\pi\ x := e_1\ \mathtt{in}\ e_2 \mid !x \mid x := e \\
\text{Types} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \tau_1 \xrightarrow{\phi} \tau_2 \mid \tau\ \mathtt{ref}\ \varrho \\
\text{Region} & \phi ::= \{!\pi\} \mid \{\pi :=\} \mid \{\mathtt{new}\ \pi\} \mid \phi_1 \cup \phi_2 \mid \varnothing \\
\text{PrgPts} & \varrho ::= \{\pi\} \mid \varrho_1 \cup \varrho_2 \mid \varnothing
\end{array}
$$

$$\Gamma \vdash e : \tau\ \&\ \phi$$

[33] F. Nielson and H. R. Nielson, "Type and Effect Systems", 1999.

# Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

# Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

# Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

# Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism
- Sound type system guarantees safety of deallocations

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

# Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism
- Sound type system guarantees safety of deallocations
- Region inference identifies:

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

# Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism
- Sound type system guarantees safety of deallocations
- Region inference identifies:
  - Points where entire regions are allocated and deallocated

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

# Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism
- Sound type system guarantees safety of deallocations
- Region inference identifies:
  - Points where entire regions are allocated and deallocated
  - Into which region values should go

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

# Region-Based Memory Managememt

Regions[34]

- Divide heap into stack of sub-heaps (i.e. regions)
- Regions grow on individual allocation; entire region deallocated
- Explicit region polymorphism
- Sound type system guarantees safety of deallocations
- Region inference identifies:
  - Points where entire regions are allocated and deallocated
  - Into which region values should go
- Unreasonable object lifetimes due to LIFO ordering of region lifetimes

[34] Tofte and Talpin, "Region-Based Memory Management", 1997.

# Early Use of Regions

Capability Calculus[35]

- Arbitrarily-ordered region allocation/deallocation, via capability tracking
- Capability: set of regions presently valid to access

[35] Crary, Walker, and G. Morrisett, "Typed Memory Management in a Calculus of Capabilities", 1999.

[36] Gay and Aiken, "Language Support for Regions", 2001.

[37] Berger, Zorn, and McKinley, "OOPSLA 2002", 2002.

[38] Kowshik, Dhurjati, and Adve, "Ensuring Code Safety Without Runtime Checks for Real-time Control Systems", 2002.

[39] Dhurjati, Kowshik, et al., "Memory safety without runtime checks or garbage collection", 2003.

# Early Use of Regions

Capability Calculus[35]

- Arbitrarily-ordered region allocation/deallocation, via capability tracking
- Capability: set of regions presently valid to access

Also see:

- RC[36]
- Reaps[37]
- Control-C[38] and Type Homogeneity[39]

[35] Crary, Walker, and G. Morrisett, "Typed Memory Management in a Calculus of Capabilities", 1999.

[36] Gay and Aiken, "Language Support for Regions", 2001.

[37] Berger, Zorn, and McKinley, "OOPSLA 2002", 2002.

[38] Kowshik, Dhurjati, and Adve, "Ensuring Code Safety Without Runtime Checks for Real-time Control Systems", 2002.

[39] Dhurjati, Kowshik, et al., "Memory safety without runtime checks or garbage collection", 2003.

## Regions in Cyclone

Cyclone,[40] again!

- Three region types:

[40] Grossman et al., "Region-based Memory Management in Cyclone", 2002.

# Regions in Cyclone

Cyclone,[40] again!

- Three region types:
  - single heap region

[40]Grossman et al., "Region-based Memory Management in Cyclone", 2002.

# Regions in Cyclone

Cyclone,[40] again!

- Three region types:
  - single heap region
  - stack regions

[40] Grossman et al., “Region-based Memory Management in Cyclone”, 2002.

# Regions in Cyclone

Cyclone,[40] again!

- Three region types:
  - single heap region
  - stack regions
  - dynamic regions

[40] Grossman et al., "Region-based Memory Management in Cyclone", 2002.

## Regions in Cyclone

Cyclone,[40] again!

- Three region types:
  - single heap region
  - stack regions
  - dynamic regions
- Lifetime subtyping: region A $<:$ region B $\Leftrightarrow$ region A outlives region B

[40] Grossman et al., "Region-based Memory Management in Cyclone", 2002.

# Regions in Cyclone

Cyclone,[40] again!

- Three region types:
  - single heap region
  - stack regions
  - dynamic regions
- Lifetime subtyping: region A $<:$ region B $\Leftrightarrow$ region A outlives region B
- Sane defaults by inferring region annotations on pointer types

[40] Grossman et al., "Region-based Memory Management in Cyclone", 2002.

## Regions in Cyclone

- Pointer can escape scope of their regions

# Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)

# Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference

# Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference
- Function effect: set of regions it might access

# Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference
- Function effect: set of regions it might access
- Calculate function's effect from prototype alone

# Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference
- Function effect: set of regions it might access
- Calculate function's effect from prototype alone

# Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference
- Function effect: set of regions it might access
- Calculate function's effect from prototype alone

```
struct Set<α, ρ, ε>
{
  list_t<α, ρ> elts;
  int (*cmp)(α,α; ε);
}
```

# Regions in Cyclone

- Pointer can escape scope of their regions
- Type system tracks capability (live regions set)
- Check needed liveness on pointer dereference
- Function effect: set of regions it might access
- Calculate function's effect from prototype alone

```
struct Set<α, ρ, ϵ>
{
  list_t<α, ρ> elts;
  int (*cmp)(α,α; ϵ);
}
```

```
struct Set<α, ρ>
{
  list_t<α, ρ> elts;
  int (*cmp)(α,α; regions_of(α));
}
```

# Some Cyclone Abstract Syntax

| | | | |
|---|---|---|---|
| kinds | $\kappa$ | $::=$ | $T \mid R$ |
| type and region vars | $\alpha, \rho$ | | |
| region sets | $\epsilon$ | $::=$ | $\emptyset \mid \alpha \mid \epsilon_1 \cup \epsilon_2$ |
| region constraints | $\gamma$ | $::=$ | $\emptyset \mid \gamma, \epsilon <: \rho$ |
| constructors | $\tau$ | $::=$ | $\alpha \mid \mathsf{int} \mid \tau_1 \xrightarrow{\epsilon} \tau_2 \mid \tau @ \rho \mid \mathsf{handle}(\rho) \mid \forall \alpha : \kappa \rhd \gamma . \tau \mid \ldots$ |
| expressions | $e$ | $::=$ | $x_\rho \mid v \mid e \langle \tau \rangle \mid * e \mid \mathbf{new}(e_1) e_2 \mid e_1(e_2) \mid \& e \mid \ldots$ |
| functions | $f$ | $::=$ | $\rho : (\tau_1\ x_\rho) \xrightarrow{\epsilon} \tau_2 = \{s\} \mid \Lambda \alpha : \kappa \rhd \gamma . f$ |
| statements | $s$ | $::=$ | $e \mid s_1 ; s_2 \mid \mathbf{if}\ (e)\ s_1\ \mathbf{else}\ s_2 \mid \rho : \{\tau\ x_\rho = e;\ s\} \mid \mathbf{region}\langle \rho \rangle\ x_\rho\ s \mid \ldots$ |

…

# Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\quad$ $\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau$ $\quad$ $\Gamma \vdash \epsilon \Rightarrow \rho$

# Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau$ $\Gamma \vdash \epsilon \Rightarrow \rho$

$\Delta$ : in-scope type/region vars

## Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau$ $\Gamma \vdash \epsilon \Rightarrow \rho$

- $\Delta$ : in-scope type/region vars
- $\Gamma$ : mapping of in-scope vars to types

## Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau$ $\Gamma \vdash \epsilon \Rightarrow \rho$

- $\Delta$ : in-scope type/region vars
- $\Gamma$ : mapping of in-scope vars to types
- $\gamma$ : constraints relating lifetimes

# Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\quad\quad$ $\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau$ $\quad\quad$ $\Gamma \vdash \epsilon \Rightarrow \rho$

$\Delta$ : in-scope type/region vars

$\Gamma$ : mapping of in-scope vars to types

$\gamma$ : constraints relating lifetimes

$\epsilon$ : capability, i.e. live regions

# Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\qquad$ $\Delta; \Gamma; \gamma; \epsilon \vdash e \ : \ \tau$ $\qquad$ $\Gamma \vdash \epsilon \Rightarrow \rho$

- $\Delta$ : in-scope type/region vars
- $\Gamma$ : mapping of in-scope vars to types
- $\gamma$ : constraints relating lifetimes
- $\epsilon$ : capability, i.e. live regions

$$\frac{\gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash x_\rho : \Gamma(x_\rho)} \ (\textsc{var}) \qquad \frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau * \rho \qquad \gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash *e : \tau} \ (\textsc{deref})$$

## Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\quad\quad$ $\Delta; \Gamma; \gamma; \epsilon \vdash e \;:\; \tau$ $\quad\quad$ $\Gamma \vdash \epsilon \Rightarrow \rho$

- $\Delta$ : in-scope type/region vars
- $\Gamma$ : mapping of in-scope vars to types
- $\gamma$ : constraints relating lifetimes
- $\epsilon$ : capability, i.e. live regions

$$\frac{\gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash x_\rho : \Gamma(x_\rho)} \text{ (VAR)} \qquad \frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau * \rho \qquad \gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash *e : \tau} \text{ (DEREF)}$$

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e_1 : \tau_2 \xrightarrow{\epsilon_1} \tau \qquad \Delta; \Gamma; \gamma; \epsilon \vdash e_2 : \tau_2 \qquad \gamma \vdash \epsilon \Rightarrow \epsilon_1}{\Delta; \Gamma; \gamma; \epsilon \vdash e_1(e_2) : \tau} \text{ (CALL)}$$

# Example Cyclone Judgments

$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{stmt} s$ $\qquad \Delta; \Gamma; \gamma; \epsilon \vdash e : \tau$ $\qquad \Gamma \vdash \epsilon \Rightarrow \rho$

- $\Delta$ : in-scope type/region vars
- $\Gamma$ : mapping of in-scope vars to types
- $\gamma$ : constraints relating lifetimes
- $\epsilon$ : capability, i.e. live regions

$$\frac{\gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash x_\rho : \Gamma(x_\rho)} \text{ (VAR)} \qquad \frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau * \rho \qquad \gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash *e : \tau} \text{ (DEREF)}$$

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e_1 : \tau_2 \xrightarrow{\epsilon_1} \tau \qquad \Delta; \Gamma; \gamma; \epsilon \vdash e_2 : \tau_2 \qquad \gamma \vdash \epsilon \Rightarrow \epsilon_1}{\Delta; \Gamma; \gamma; \epsilon \vdash e_1(e_2) : \tau} \text{ (CALL)}$$

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \forall \alpha : \kappa \rhd \gamma_1.\tau_2 \qquad \Delta \vdash \tau : \kappa \qquad \gamma \vdash \gamma_1[\tau_1/\alpha]}{\Delta; \Gamma; \gamma; \epsilon \vdash e\langle \tau_1 \rangle : \tau_2[\tau_1/\alpha]} \text{ (TYPE-INST)}$$

## Cyclone Results

Soundness Theorem[41]:

- The program cannot get stuck from type errors or dangling-pointer dereferences.
- The terminating program deallocates all regions it allocates

[41] Grossman et al., *Formal Type Soundness for Cyclone's Region System*, 2001.

# Cyclone Results

Soundness Theorem[41]:

- The program cannot get stuck from type errors or dangling-pointer dereferences.
- The terminating program deallocates all regions it allocates

Benchmarks

- 86 lines of region annotation-related changes across 18,000 lines (6%)
- Eliminate heap allocation entirely for web-server
- Near-zero overhead (from garbage collection and bounds checking)

[41] Grossman et al., *Formal Type Soundness for Cyclone's Region System*, 2001.

# Outline

## Linear Types

Linear types[42,43] ensure that every variable is used exactly *once*.

- The world is a non-duplicatable resource

[42] Girard, "Linear logic", 1987.

[43] Wadler, "Linear types can change the world", 1990.

# Linear Types

Linear types[42,43] ensure that every variable is used exactly *once*.

- The world is a non-duplicatable resource
- Track proper modification of world

[42] Girard, "Linear logic", 1987.

[43] Wadler, "Linear types can change the world", 1990.

## Linear Types

Linear types[42,43] ensure that every variable is used exactly *once*.

- The world is a non-duplicatable resource
- Track proper modification of world
- Efficiency: safe to destructively update an array

[42] Girard, "Linear logic", 1987.

[43] Wadler, "Linear types can change the world", 1990.

# Linear Types

Linear types[42,43] ensure that every variable is used exactly *once*.

- The world is a non-duplicatable resource
- Track proper modification of world
- Efficiency: safe to destructively update an array
- **Memory management: can immediately collect used values**

[42]Girard, "Linear logic", 1987.

[43]Wadler, "Linear types can change the world", 1990.

# Applied Linear Types

Clay[44]

- Type-theoretic basis for giving type-safe code more control over memory
- Singleton types to type check loads, coercion functions to modify values' type safely

[44] Hawblitzel et al., "Low-Level Linear Memory Management", 2004.

[45] Ennals, Sharp, and Mycroft, "Linear Types for Packet Processing", 2004.

# Applied Linear Types

Clay[44]

- Type-theoretic basis for giving type-safe code more control over memory
- Singleton types to type check loads, coercion functions to modify values' type safely

PACLANG[45]

- Program network processors for handling packets
- Unique ownership property: each packet in heap is referenced by exactly one thread
- Allow mutable aliasing within the same thread
- Operations for a functions to 1) take ownership or 2) create local aliases

[44] Hawblitzel et al., "Low-Level Linear Memory Management", 2004.

[45] Ennals, Sharp, and Mycroft, "Linear Types for Packet Processing", 2004.

## COGENT

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code

[46] Amani et al., "Cogent", 2016.

[47] O'Connor et al., "COGENT", 2016.

# COGENT

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates

[46]Amani et al., "Cogent", 2016.

[47]O'Connor et al., "COGENT", 2016.

# COGENT

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:

[46] Amani et al., "Cogent", 2016.

[47] O'Connor et al., "COGENT", 2016.

## COGENT

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:
    - Ensure safe handling of heap-allocated objects

[46] Amani et al., "Cogent", 2016.
[47] O'Connor et al., "COGENT", 2016.

# COGENT

COGENT[46, 47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:
  - Ensure safe handling of heap-allocated objects
  - Equational functional semantics via mutable state/imperative effects

[46] Amani et al., "Cogent", 2016.

[47] O'Connor et al., "COGENT", 2016.

# COGENT

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:
  - Ensure safe handling of heap-allocated objects
  - Equational functional semantics via mutable state/imperative effects
- Reason with interactive theorem prover

[46] Amani et al., “Cogent”, 2016.

[47] O’Connor et al., “COGENT”, 2016.

# COGENT

COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:
  - Ensure safe handling of heap-allocated objects
  - Equational functional semantics via mutable state/imperative effects
- Reason with interactive theorem prover
- Missing functionality can be implemented in C, manually verified

[46] Amani et al., "Cogent", 2016.
[47] O'Connor et al., "COGENT", 2016.

# COGENT

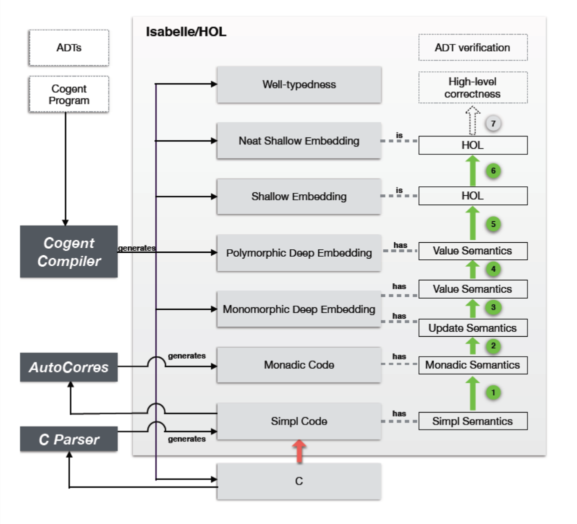
COGENT[46,47]

- Pure, polymorphic language with linear types for writing low-level systems code
- Efficient machine code with in-place updates
- Linear types:
  - Ensure safe handling of heap-allocated objects
  - Equational functional semantics via mutable state/imperative effects
- Reason with interactive theorem prover
- Missing functionality can be implemented in C, manually verified
- No trusted compiler, runtime, or garbage collector needed

[46] Amani et al., "Cogent", 2016.

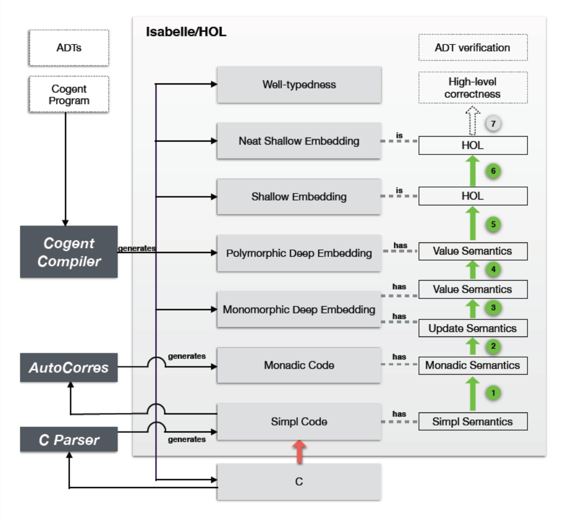
[47] O'Connor et al., "COGENT", 2016.

# COGENT



ADTs
Cogent Program
Isabelle/HOL
ADT verification
High-level correctness
Well-typedness
Neat Shallow Embedding
Shallow Embedding
Polymorphic Deep Embedding
Monomorphic Deep Embedding
Monadic Code
Simpl Code
C
Cogent Compiler
AutoCorres
C Parser
generates
is
has
HOL
Value Semantics
Update Semantics
Monadic Semantics
Simpl Semantics
1
2
3
4
5
6
7

# COGENT




Person-Months of Work

- Proof Framework: 33.5
- Compiler: 10
- Proofs: 18

Lines (kLOC)

- Isabelle theorems: 17
- Compiler: 9.5
- ext2 Filesystem: 6.5 (Isabelle/HOL: 76.7)

# Other Linear Types

Quasi-linear types[48]

- Distinguish consumed values from those that may be returned
- Use $\kappa$ to control how often a variable of type $\tau^\kappa$ is used (many times locally)

[48] Kobayashi, "Quasi-linear Types", 1999.

[49] DeLine and Fähndrich, "Enforcing High-level Protocols in Low-level Software", 2001.

[50] Petersen et al., "A Type Theory for Memory Allocation and Data Layout", 2003.

# Other Linear Types

Quasi-linear types[48]

- Distinguish consumed values from those that may be returned
- Use $\kappa$ to control how often a variable of type $\tau^\kappa$ is used (many times locally)

Vault[49]

- Keys associate static capabilities with run-time resources
- Annotate functions with effect clause (pre- and post-conditions on held-key set)
- Windows 2000 locking errors, IRP ownership model

[48] Kobayashi, "Quasi-linear Types", 1999.

[49] DeLine and Fähndrich, "Enforcing High-level Protocols in Low-level Software", 2001.

[50] Petersen et al., "A Type Theory for Memory Allocation and Data Layout", 2003.

# Other Linear Types

Quasi-linear types[48]

- Distinguish consumed values from those that may be returned
- Use $\kappa$ to control how often a variable of type $\tau^\kappa$ is used (many times locally)

Vault[49]

- Keys associate static capabilities with run-time resources
- Annotate functions with effect clause (pre- and post-conditions on held-key set)
- Windows 2000 locking errors, IRP ownership model

Ordered types for memory layout[50]

- Variables must be used in order $\Rightarrow$ memory locations
- *Orderly lambda calculus* for size-preserving memory operations

[48] Kobayashi, "Quasi-linear Types", 1999.

[49] DeLine and Fähndrich, "Enforcing High-level Protocols in Low-level Software", 2001.

[50] Petersen et al., "A Type Theory for Memory Allocation and Data Layout", 2003.

# Ownership

Types can represent ownership and prevent *aliasing* and *mutation* on the same location.

---

[51] Evans, "Static Detection of Dynamic Memory Errors", 1996.

[52] Clarke, Potter, and Noble, "Ownership Types for Flexible Alias Protection", 1998.

[53] Fähndrich et al., "Language Support for Fast and Reliable Message-based Communication in Singularity OS", 2006.

# Ownership

Types can represent ownership and prevent *aliasing* and *mutation* on the same location.

LCL[51]

- `owned` annotation to denote reference with obligation to release storage
- `dependent` annotation for sharing; user ensures lifetimes contained properly

[51] Evans, "Static Detection of Dynamic Memory Errors", 1996.

[52] Clarke, Potter, and Noble, "Ownership Types for Flexible Alias Protection", 1998.

[53] Fähndrich et al., "Language Support for Fast and Reliable Message-based Communication in Singularity OS", 2006.

# Ownership

Types can represent ownership and prevent *aliasing* and *mutation* on the same location.

LCL[51]

- `owned` annotation to denote reference with obligation to release storage
- `dependent` annotation for sharing; user ensures lifetimes contained properly

Ownership Types[52]

- Object's definition includes unique object context that owns it

[51] Evans, "Static Detection of Dynamic Memory Errors", 1996.

[52] Clarke, Potter, and Noble, "Ownership Types for Flexible Alias Protection", 1998.

[53] Fähndrich et al., "Language Support for Fast and Reliable Message-based Communication in Singularity OS", 2006.

# Ownership

Types can represent ownership and prevent *aliasing* and *mutation* on the same location.

LCL[51]

- `owned` annotation to denote reference with obligation to release storage
- `dependent` annotation for sharing; user ensures lifetimes contained properly

Ownership Types[52]

- Object's definition includes unique object context that owns it

Singularity[53]

- Type system tracks resources, passes ownership of arguments to callee

---

[51] Evans, "Static Detection of Dynamic Memory Errors", 1996.

[52] Clarke, Potter, and Noble, "Ownership Types for Flexible Alias Protection", 1998.

[53] Fähndrich et al., "Language Support for Fast and Reliable Message-based Communication in Singularity OS", 2006.

# Rust[54,55,56]

- Ownership and lifetimes

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner

[54]Matsakis and Klock, "The Rust Language", 2014.
[55]Levy et al., "Ownership is Theft", 2015.
[56]Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions $\approx$ lifetimes (region capabilities $\approx$ lifetime tokens)

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions $\approx$ lifetimes (region capabilities $\approx$ lifetime tokens)

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
        let mut v = Vec::new();
        v.push(0);
        snd.send(v);
        v.push(1);
    },
    move || {
        let v = rcv.recv().unwrap();
        println!("Received:␣{:?}", v);
    });
```

[54] Matsakis and Klock, "The Rust Language", 2014.

[55] Levy et al., "Ownership is Theft", 2015.

[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
      let mut v = Vec::new();
      v.push(0);
      snd.send(v);
      v.push(1);
    },
    move || {
      let v = rcv.recv().unwrap();
      println!("Received:␣{:?}", v);
    });
```

[54] Matsakis and Klock, "The Rust Language", 2014.

[55] Levy et al., "Ownership is Theft", 2015.

[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
      let mut v = Vec::new();
      v.push(0);
      snd.send(v);
      v.push(1);
    },
    move || {
      let v = rcv.recv().unwrap();
      println!("Received: {:?}", v);
    });
```

[54] Matsakis and Klock, "The Rust Language", 2014.

[55] Levy et al., "Ownership is Theft", 2015.

[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
      let mut v = Vec::new();
      v.push(0);
      snd.send(v);
      v.push(1);
    },
    move || {
      let v = rcv.recv().unwrap();
      println!("Received: {:?}", v);
    });
```

[54] Matsakis and Klock, "The Rust Language", 2014.

[55] Levy et al., "Ownership is Theft", 2015.

[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
      let mut v = Vec::new();
      v.push(0);
      snd.send(v);
      v.push(1);
    },
    move || {
      let v = rcv.recv().unwrap();
      println!("Received:␣{:?}", v);
    });
```

[54] Matsakis and Klock, “The Rust Language”, 2014.

[55] Levy et al., “Ownership is Theft”, 2015.

[56] Jung et al., “RustBelt”, 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
      let mut v = Vec::new();
      v.push(0);
      snd.send(v);
      v.push(1);
    },
    move || {
      let v = rcv.recv().unwrap();
      println!("Received:␣{:?}", v);
    });
```

[54] Matsakis and Klock, "The Rust Language", 2014.
[55] Levy et al., "Ownership is Theft", 2015.
[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
        let mut v = Vec::new();
        v.push(0);
        snd.send(v);
        v.push(1);
    },
    move || {
        let v = rcv.recv().unwrap();
        println!("Received:␣{:?}", v);
    });
```

[54] Matsakis and Klock, "The Rust Language", 2014.

[55] Levy et al., "Ownership is Theft", 2015.

[56] Jung et al., "RustBelt", 2017.

# Rust[54,55,56]

- Ownership and lifetimes
- Type system enforces that objects have unique owners
- Objects may be borrowed for no longer than owner
- Object automatically deallocated when owner leaves scope
- Unsafe sections for mutating raw pointers and aliased state
- Regions ≈ lifetimes (region capabilities ≈ lifetime tokens)

```
let (snd, rcv) = channel();
join(move || {
      let mut v = Vec::new();
      v.push(0);
      snd.send(v);
      v.push(1);
    },
    move || {
      let v = rcv.recv().unwrap();
      println!("Received: {:?}", v);
    });
```

[54] Matsakis and Klock, "The Rust Language", 2014.

[55] Levy et al., "Ownership is Theft", 2015.

[56] Jung et al., "RustBelt", 2017.

# Quick Temporal Recap

- Capabilities and pointer-based metadata

# Quick Temporal Recap

- Capabilities and pointer-based metadata
- Effects and regions

# Quick Temporal Recap

- Capabilities and pointer-based metadata
- Effects and regions
- Linear types

# Quick Temporal Recap

- Capabilities and pointer-based metadata
- Effects and regions
- Linear types
- Ownership and borrowing

# Conclusion

- Memory errors $\equiv$ *type errors*

# Conclusion

- Memory errors $\equiv$ *type errors*
- *Static* memory management

# Conclusion

- Memory errors $\equiv$ *type errors*
- *Static* memory management
- Isolate unsafe world

# Conclusion

- Memory errors $\equiv$ *type errors*
- *Static* memory management
- Isolate unsafe world
- Be reasonable and optimistic

# Thanks

Thanks!

# Referent Objects Approaches

J&K[57]

- Maintain table of all known valid storage objects
- Map a ptr to a desciptor of the object into which it points
- Pad objects w/ extra byte
- check ptr arith and use, b/c result cannot refer to diff object from one from which it is originally derived
- object for which the ptr is valid is only determined by checking ptr itself, looking it up in object table
- Incomplete because cannot pad parameters, would change layout
- 11-12x overhead

CRED[58]

- More generic solution
- 2x overhead

[57] Jones and Kelly, “Backwards-compatible bounds checking for arrays and pointers in C programs”, 1997.

# BBC

Baggy Bounds Checking (BBC)[59]

- Trades memory for performance, fastest object bounds checker
- Compact bounds representation and efficient way to look up object bounds
- Align base addresses to be multiple of padded size
- Replace splay tree with small lookup table
- 0.6x overhead on SPECINT 2000 benchmark
- Partition memory into slots with slot_size bytes (table has entry per slot rather than per byte)
- Pad every object s.t. size is power of two, align base addresses to be multiple of their padded size
- Mark if OOB to prevent later dereference
- Handle OOB withint slot_size/2 bytes from original object

---

[59]Akritidis et al., "Baggy Bounds Checking", 2009.

# Other Spatial Approaches

- Purify[60] (reveal during testing, object based, has temporal safety)
- Valgrind[61] (reveal during testing)

[60] Hastings and Joyce, "Purify", 1991.

[61] Nethercote and Seward, "How to shadow every byte of memory used by a program", 2007.

# Intel MPX

Intel MPX

- ”new instructions that a compiler can insert to accelerate disjoint per-pointer metadata access and bounds checking”
- 4 new 128-bit bound registers
- BNDCL, BNDCU
- BNDSTC, BNDLDX
- BNDMK – create bounds metadata for a pointer
- Extends func call conventions to include these bound registers
- Disjoint metadata for ptrs in memory
- Does not address temporal errors
- ’Incremental deployment’ by checking if loaded pointer is same as redundantly-stored ptr in disjoint metadata structure => loss of comprehensiveness
- Metadata isn’t updated on *integer stores* (b/c don’t know to treat it as a ptr)

# Alias Types

Alias types[62]

- Problem: registers must be reused for data of different types constantly
- Aliasing constraints: describe shape of store; functions use to specify what they expect part of the store to look like
- Location and store polymorphism: dependence between pointer types and constraints, abstract away size/shape of store
- More expressive than linear: although aliasing constraints are linear, ptr values that flow through computation are not
- Constraint is mapping from locations to types; Pointer to location l has singleton type ptr(l)

[62] Smith, Walker, and J. G. Morrisett, "Alias Types", 2000.

# More Object-Based Temporal Safety

- Mark locations which were de-allocated in shadow memory space (i.e. track a few bits of state for each byte in memory, indicating if location is currently valid)
- Detect access of de-allocated locations
- Fails if pointer points again to re-allocated space
- Memcheck (10x slowdown)
- Address Sanitizer[63]
  - Tripwire approach
  - 73% slowdown
  - Can detect small-stride buffer overflows

[63] Serebryany et al., "AddressSanitizer: A Fast Address Sanity Checker", 2012.

# Extended Static Checking

Extended Static Checking (ESC)[64]

- Use automatic theorem prover to detect index bounds in Modula-3
- Use info in annotations to assist
- Easier than C b/c no ptr arithmetic

Also see: ESC/Java[65]

[64] Detlefs, "An Overview of the Extended Static Checking System", 1995.

[65] Flanagan et al., "Extended Static Checking for Java", 2002.

# LCLint

LCLint[66]

- Leverage LCLint, an annotation-assisted buffer detection tool
- Annotations that constrain possible values a reference contains before/after funcall
- Function pre/post-condition with:

  `requires`, `ensures`, `unique`, `returned`, `modifies`, `out` clauses
- Assumptions are `minSet`, `maxSet`, `minRead`, `maxRead`
- Generates constraints at expression level, resolved w/ checking at statement level
- Heuristics to deal with loops nicely enough; neither sound nor complete

Also see: ESC/Modula-3 ,[67] ESC/Java[68]

[66] Larochelle and Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities", 2001.

[67] Detlefs, "An Overview of the Extended Static Checking System", 1995.

[68] Flanagan et al., "Extended Static Checking for Java", 2002.

# CSSV

CSSV[69]

- Source-to-source translation
- Instruments program w/ additional variables describing string attrs
- Adds assert statements checking for unsafe string ops
- Statically analyze instr. version with *integer analysis* to determine possible assertion failures
- Handles overlapping ptrs, etc.
- Disadv: # vars in instr. quadratic in # in orig.

[69] Dor, Rodeh, and Sagiv, "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C", 2003.

# Early Use of Regions

RC[70]

- Region-support for C, focus on preventing dangling pointers
- No restrictions on region lifetimes, pointers can point anywhere
- Maintain a reference count of external pointers pointing to objects in region
- Type system + annotations (`sameregion`, `traditional`, `parentptr`) to help compiler remove annotation operations
- Annotations (`sameregion`, `traditional`, `parentptr`) to prevent updating associated region
- Restrict C by disallowing arbitrary integer-to-pointer casts
- Break cyclic data structures before deleting regions
- No memory safety guarantee
- 27%-11% overhead

[70] Gay and Aiken, "Language Support for Regions", 2001.

# More Early Use of Regions

Reaps[71]

- Combine regions and heaps into "reaps"; regions until `reapFree` deallocates individual object
- Place freed object on a heap; subsequent allocations use heap until exhausted
- Use Lea allocator generally, unless you need fast regions (i.e. reaps)

Control-C[72]

- Single region active at any given time
- Pointer value containing region address must be provably dead before `rfree`
- Automatic Pool Allocation and Type Homogeneity[73] for safe dangling pointers

[71] Berger, Zorn, and McKinley, "OOPSLA 2002", 2002.

[72] Kowshik, Dhurjati, and Adve, "Ensuring Code Safety Without Runtime Checks for Real-time Control Systems", 2002.

[73] Dhurjati, Kowshik, et al., "Memory safety without runtime checks or garbage collection", 2003.

# CETS

CETS

- Extends SoftBound
- Validity bit stored in global dictionary
- Each objects gets unique id as key to dictionary, pointers associated with id
- enforces full safety
- 48% for temporal, 116% overhead for full (Softbound-CETS) on SPEC
- incompatiblity issues of SoftBound too

# Softbound-CETS

- In absence of wrappers, code will not experience vilations as long as external libaries d no return ptrs or update ptrs in memory
- Benefits from LLVM maintaing ptr info from src in IR: large # of checks can be eliminated statically using check elim opt
- 76% overhead on avg on SPEC benchmarks

# Memsafe

Memsafe[74]

- Use spatial data for temporal safety by setting bounds of deallocated ptr and all its aliased ptrs in metadata space to an invalid value;
- Subsequent deference of such a deallocated ptr will raise an exception as bounds metadata is invalid
- Also stores some object-based metadata in global database, looking up only when ptr-based meatadata is insufficient for proving temporal safety
- Free(p) => p=invalid (i.e. addr_p->base=1 addr_p->bound=0)
- Spatial safety checks involving the base/bound of invalid ptr are guaranteed to always report a safety violation
- Inserts assignments of invalid ptr at end of a procedure for each local variable whose address is taken (so ptr to stack object can't escape)
- Models in-memory ptr assignments as epxlicit assignments using alias analysis and $\phi$-like ssa extension call $\rho$2-function

[74] Simpson and Barua, "MemSafe", 2013.

# Regions Example

```
// Example prototypes
char?ρ1 strcpy<ρ1,ρ2>(char?ρ1 d,
  const char ?ρ2 s);
char?ρH strdup<ρ>(const char?ρ s);
char?ρ strlen<ρ>(const char?ρ s);

// Dangling pointer prevention
int *ρL p;
L:{ int x = 0;
    p = &x;
  }
*p = 42;
```

```
// Polymorphic Recursion
void fact<ρ>(int*ρ result, int n) {
 L: { int x = 1;
      if (n > 1) fact<ρL>(\&x,n-1);
      *result = x*n; }
}
int g = 0;
int main() { fact<ρH>(\&g, 6); return g
 ; }
```

# Quasi-Linear Types

Quasi-linear types[75]

- Relax linear type strong condition
- Distinguish between consumed values vs those that may be returned
- Use $\kappa$ to control how often a variable of type $\tau^{\kappa}$ is used
- $\kappa = \delta$: accessed many times *locally*, cannot be returned
    - 0: not used at all
    - 1: value accessed at most once
    - $\omega$: accessed arbitrary number of times
    - $\delta$: accessed many times *locally*, cannot be returned with result
- Quasi-linear value (1) accessed as $\delta$ and then strictly as linear
- Inspired PACLANG

---

[75] Kobayashi, "Quasi-linear Types", 1999.

# Other Linear Types

Vault[76]

- Keys associate static capabilities with run-time resources
- Functions annotated with effect clause (pre- and post-conditions on held-key set contents)
- Freed regions before leaving scope
- Types enforce code must free a region
- static enforcement of various protocols
- Restrict aliasing, tracks fine-grained effects (requires more annotations)
- Windows 2000 locking errors, IRP ownership model

[76] DeLine and Fähndrich, "Enforcing High-level Protocols in Low-level Software", 2001.

# Ordered Types

Ordered types for memory layout[77]

- Restrict linear types (remove exchange property)
- Variables cannot change position $\Rightarrow$ locations in memory
- "Orderly lambda calculus" for size-preserving memory operations
- Coercions to manipulate ordered variables in frontier (combine/split to treat as different types)

[77] Petersen et al., "A Type Theory for Memory Allocation and Data Layout", 2003.

# Typestate

Typestate[78]

- Avoid nonsensical execution sequences statically (using uninitialized value)
- Typestate is static invariant of each variable name at program point
- Define a lattice of states and typestate transition system between them
- Linear types help because of restricting pointer assignment (1-1 mapping between variable names and run-time objects)

[78] Strom and Yemini, "Typestate", 1986.

# Some Practical Affine Types

Alms[79]

- Practical and general purpose
- Affine types: a *weakening* of linear types: can drop but not duplicate
- Affine capabilities: separate a read-only reference to array from an affine writeable reference
- Define ${}^{a}\lambda_{ms}$ (based on System $F^{\omega}_{<:}$) and proof of soundness
- Implemented a capability-based interface to Berkeley sockets
- Basis of Rust's type system

[79] Tov and Pucella, "Practical Affine Types", 2011.

# Low-Level Liquid Types

Low-Level Liquid Types (LTLL)[80]

- Refinement types where predicates are conjunctions over qualifiers
- Functions qualified over locations they operate on
- Deal with collections using *location folding* for checking out a copy to do strong updates on
- Tries to deal with lack of types, mutation, unbounded collections that make type-based mechanisms difficult

[80] Rondon, Kawaguchi, and Jhala, "Low-level Liquid Types", 2010.

# Cyclone

Cyclone[81]:

- Annotations for non-array vs array pointers (can specify size)
- Tagged unions and automatic tag injection
- Need user annotations more than other approaches
- 40 percent runtime overhead
- Uses regions + automatic memory management for temporal safety (free is a no-op) (see nice example)
- Never null don't need checks, use @; push back null checks from uses to their sources
- Restrict arithmetic on regular pointers

[81] Jim et al., "Cyclone: A Safe Dialect of C.", 2002.

# All Cyclone's Abstract Syntax

| | | |
|---|---|---|
| kinds | $\kappa$ | $::= T \mid R$ |
| type and region vars | $\alpha, \rho$ | |
| region sets | $\epsilon$ | $::= \emptyset \mid \alpha_n \epsilon_1 \cup \epsilon_2$ |
| region constraints | $\gamma$ | $::= \emptyset \mid \gamma, \epsilon <: \rho$ |
| constructors | $\tau$ | $::= \alpha \mid \mathsf{int} \mid \tau_1 \xrightarrow{\epsilon} \tau_2 \mid \tau_1 \times \tau_2 \mid \tau * \rho \mid$ |
| | | $\mathsf{handle}(\rho) \mid \forall \alpha : \kappa \rhd \gamma . \tau \mid \exists \alpha : \kappa \rhd \gamma . \tau$ |
| expressions | $e$ | $::= x_\rho \mid v \mid e \langle \tau \rangle \mid (e_1, e_2) \mid e.i \mid\ * e \mid \mathbf{rnew}(e_1) e_2 \mid$ |
| | | $e_1(e_2) \mid \&e \mid e_1 = e_2 \mid \mathbf{pack}[\tau_1, e]\ \mathbf{as}\ \tau_1$ |
| values | $v$ | $::= i \mid f \mid \&p \mid \mathbf{region}(\rho) \mid (v_1, v_2) \mid \mathbf{pack}[\tau_1, v]\ \mathbf{as}\ \tau_2$ |
| paths | $p$ | $::= x_\rho \mid p.i$ |
| functions | $f$ | $::= \rho : (\tau_1\ x_\rho) \xrightarrow{\epsilon} \tau_2 = \{s\} \mid \Lambda \alpha : \kappa \rhd \gamma . f$ |
| statements | $s$ | $::= e \mid \mathbf{return}\ e \mid s_1 ; s_2 \mid \mathbf{if}\ (e)\ s_1\ \mathbf{else}\ s_2 \mid \mathbf{while}\ (e)\ s \mid$ |
| | | $\rho : \{\tau\ x_\rho = e;\ s\} \mid \mathbf{region}\langle \rho \rangle\ x_\rho\ s \mid \rho : \{\mathbf{open}[\alpha, x_\rho] = e;\ s\} \mid s\ \mathbf{pop}[\rho]$ |

# Fail-Safe C

- Safe implementation of ANSI-C, handles casts very well
- Extend fat pointers to also contain
  - 'Cast' flag embedded in two-word representation: enables optimization by not needing to do a long memory access via reading header information elsewhere
  - 'Virtual Offsets' instead of real memory address offsets
- Fat Integer: integer large enough to hold any pointer value
- Virtual offset: corresponds to program-visible size
- Virtual size: real size of equivalent data type in native C impl.

# Fat Pointers

Cuckoo[82]

- Store array size in memory before array dimensions' first element
- Name of an array is pointer to an *array*, not first object
- Type system for preventing assignment of automatic objects into longer-lifetime pointers
- Wrap dynamic memory allocation (type homogeneous pool-based)
- Forbid addition and subtraction expressions including pointer operands
- Compile-time checks if array bounds are constants, otherwise run-time checks

---

[82] West and Wong, "Cuckoo", 2005.

# Hardware and Other Support for Spatial Safety

- Hardbound[83]
- Stackguard[84] (inserts canaries)
- Light-weight Bounds Checking[85] (guard zones with good performance)
- Manual MM to high-level languages[86]
- PCC[87]
- BitC[88] (a retrospective on BitC, what they were looking for (not too interesting))
- Intel MPX
- Watchdog: Nagarakatte 2012
- (Hybrid): WatchdogLite

[83] Devietti et al., "Hardbound", 2008.

[84] Cowan et al., "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks", 1998.

[85] Hasabnis, Misra, and Sekar, "Light-weight Bounds Checking", 2012.

[86] Kedia et al., "Simple, fast, and safe manual memory management", 2017.

[87] Necula, "Proof-carrying Code", 1997.

[88] Shapiro, Sridhar, and Doerrie, *Warning*, 2008.

# Rust OS

Rust OS[89]

- Ownership *hinders* resource sharing
- AMM not optimized for HW resources/device drivers
- Closures' req for dynamic memory is bad for embedded systems
- Many resources not dynamically allocated
- Mutably borrow static resources
- Embedded systems typically have one primary execution thread, so aliasing in same thread is okay
- Rust doesn't allow mutable aliasing, so extend type system with execution contexts
- Type records thread of value's owner: allow multiple borrows of value within same thread, not across threads

[89] Levy et al., "Ownership is Theft", 2015.

## No Time

- Mudflap: Eigler 2003
- Criswell: Secure Virt Arch 2007
- PariCheck: Younan: 2010