Credit where credit is due...

- Cyclone is a research language, the product of the labors of many people:
  - Greg Morrisett (Harvard)
  - Dan Grossman (Washington)
  - Trevor Jim (AT&T)
  - Mike Hicks

Other Useful Papers

- Region-based Memory Management in Cyclone (PLDI 02)
- Experience with Safe Manual Memory Management in Cyclone (ISMM 04)
- See the Cyclone web page.

1988? 2004?

- “In order to start copies of itself running on other machines, the worm took advantage of a buffer overrun…
  ...it is estimated that it infected and crippled 5 to 10 percent of the machines on the Internet.”

  Fact: half of CERT advisories involve buffer overruns.

1998: Missile Cruisers

- “The controversy began when the USS Yorktown … suffered a widespread system failure … a crew member mistakenly entered a zero into the data field of an application … caused a buffer overflow … which turned into a memory leak … eventually brought down the ship’s propulsion system.
  The result: the Yorktown was dead in the water for more than two hours.”

Building Secure Software

- Today, our economy, government, and military depend upon the proper functioning of our computing and communications infrastructure.
- That infrastructure is coded in low-level, error-prone languages (i.e. C).
  - device drivers, kernels
  - file systems, web servers, email systems
  - switches, routers, firewalls
But C is a lousy language

- Must bypass the type system to do even simple things (e.g., allocate and initialize an object.)
- Libraries put the onus on the programmer to do the “right thing” (e.g., check return codes, pass in large enough buffer.)
- For efficiency, programmers stack-allocate arrays of size K (is K big enough? does the array escape downwards?)
- Programmers assume objects can be safely recycled when they cannot and fail to recycle memory when they should.
- It’s not “fail-stop” --- errors don’t manifest themselves until well after they happen (e.g., buffer overruns.)

But it’s also very useful:

- Almost every critical system is coded in C:
  - language run-times, operating systems, device drivers, servers, switches, etc.
  - because it provides a lot of good things:
    - ported to lots of architectures
    - low-level control over data structures, memory management, instructions, etc.
    - good performance
- We need safety for these infrastructures.

What can we do?

- Rewrite the code in Java or some other type-safe language?
  - Not low-level enough.
    - no control over data representations.
    - no control over memory management.
    - performance isn’t there?
  - Just not realistic.
    - any more than telling all of those businesses to re-code their Cobol code to avoid Y2K.
    - need an incremental solution.

Instead ...

- We need a next-generation low-level language X with the following features:
  - The practical coding power of C.
    - need to build device drivers, kernels, etc.
  - Transparent interoperability with legacy C.
    - just can’t switch the whole world over at once.
  - The safety and scalability of Java.
    - many errors caught at compile time
    - fail-stop behavior at run time.
  - A relatively painless path from C to X.

Cyclone: an experimental Safe-C

- Start with ANSI-C.
- Throw out anything that can lead to a delayed core-dump:
  - e.g., arbitrary casts, unchecked pointer arithmetic
- Add a combination of advanced typing mechanisms and dynamic checks to cover what’s missing.
  - keep analyses intra-procedural.
  - programmer will have to specify additional details at procedure boundaries.
- Minimize re-coding for safe idioms.
  - best case: leave the code alone
  - next best: add typing annotations
  - worst case: re-write the code

What is a C buffer overflow?

```
#include <stdio>

int login() {
    char user [100];
    printf("login: ");
    scanf("%s", &user);
    // get password etc.
}
```

What happens if the user types something that’s more than 100 characters?
Calling scanf()  

Stack grows downward  

32 bits  

int login() {  
    char user[100];  
    printf("login: ");  
    scanf("%s", &user);  
    // return here  
}  

Calling scanf()  

Stack grows downward  

32 bits  

int login() {  
    char user[100];  
    printf("login: ");  
    scanf("%s", &user);  
    // return here  
}  

Calling scanf()  

Stack grows downward  

32 bits  

int login() {  
    char user[100];  
    printf("login: ");  
    scanf("%s", &user);  
    // return here  
}  

Calling scanf()  

Stack grows downward  

32 bits  

int login() {  
    char user[100];  
    printf("login: ");  
    scanf("%s", &user);  
    // return here  
}  

Calling scanf()  

Stack grows downward  

32 bits  

int login() {  
    char user[100];  
    printf("login: ");  
    scanf("%s", &user);  
    // return here  
}  

How to Prevent This?  

- Don’t allow dereferencing a buffer unless compiler can prove it’s safe  
  - Too conservative  
- Have two separate stacks, one for data, one for return addresses  
  - Violates standard calling convention  
  - Could still work around this  
- Prevent dereferencing with dynamic checks  

Bounds Checking  

- I would like scanf to check each time it writes to its buffer to make sure that it’s not about to “go off the end.”  
- To do this, I must provide not only the buffer memory, but the bounds on it.  
- Then I can check that every dereference is within bounds.  
- This is what Java does, too.
“Fat” pointers

• What kind of bounds do I need?
  - Just the length of the array
    • This is what Java does
      • But, what happens with pointer arithmetic?
  - A pointer to the current location, and a pointer to the end of the array
    • Allows forward arithmetic (x++)
      • But what about backward arithmetic (x--)
    • Answer: pointers to the beginning and end of the buffer, and a pointer to the current location.

“Fat” pointer implementation

A “thin” pointer (one word)
Pointer arithmetic unsafe

p

q

q++;
q++;
q++;

A “fat” pointer (three words)
Pointer arithmetic OK

H E L L O

q

“Fat” pointer implementation

A “thin” pointer (one word)
Pointer arithmetic unsafe

p

q

q++;
q++;
q++;

A “fat” pointer (three words)
Pointer arithmetic OK

H E L L O

q

“Fat” pointer implementation

A “thin” pointer (one word)
Pointer arithmetic unsafe

p

q

q++;
q++;
q++;

A “fat” pointer (three words)
Pointer arithmetic OK

H E L L O

q

“Fat” pointer implementation

A “thin” pointer (one word)
Pointer arithmetic unsafe

p

q

q++;
q++;
q++;

A “fat” pointer (three words)
Pointer arithmetic OK

H E L L O

q

“Fat” pointer implementation

A “thin” pointer (one word)
Pointer arithmetic unsafe

p

q

q++;
q++;
q++;

A “fat” pointer (three words)
Pointer arithmetic OK

H E L L O

q
Thin, bounded pointers

```c
#include <stdio>

int foo() {
    char buf[100] = {'h', 'e', 'l', 'l', '
    for (i = 0; i<100; i++) {
        putc(buf[i]);
    }
}
```

Do I really need bounds checks here?

No. Compiler can easily prove that all dereferences will be in bounds, so no need for extra information.

What about NULL?

```c
#include <stdio>

int foo(char *filename, char *buf) {
    FILE *fp;
    fp = fopen(filename, "r");
    fwrite(fp, buf);
}
```

What happens if `fopen` failed, returning NULL?

Can result in a crash. C library assumes the user will check for NULL. In Cyclone we enforce this.

Not-null Pointers

- Two pointer types
  - `int *`  
    - A possibly-null pointer to an `int`
  - `int *@notnull`  
    - A definitely-not-null pointer to an `int`
    - Abbreviated `int` @
- Library functions can specify the latter, thus forcing the user to do a null check.

Not-null Pointer Usage

```c
int *p = NULL;
int @q = NULL; // not allowed
int @r = p; // not allowed; type(p) != type(r)
int @r = (int @)p; // ok, does a null check
extern int fwrite(FILE @fp, char *buf);
// requires that fp be not-null
```

Pointer Summary

- Three kinds of pointers make intention clear:
  - fat pointers: `int ?`
    - represented as a triple: {base, upper, curr}
    - supports all operations that C does on `int`
    - but any dereference is checked against bounds
    - ? makes representation change clear
  - thin, definite pointers: `int @`, `int @[const-exp]`
    - thin, possibly null pointers: `int *`, `int *[const-exp]`
    - bounds tracked statically -- same rep. as C
    - limited pointer arithmetic
    - * requires a null check.

Cyclone Hello World

```c
#include <stdio.h>

int main(int argc, char **argv) {
    if (argc > 1) {
        printf("Hello %s\n", argv[1]);
        return 0;
    }
    fprintf(stderr, "Usage: %s <name>\n", argv[0]);
    return -1;
}
```

Libraries are wrapped to prevent bad inputs

* denotes a "fat" pointer with bounds information

Arguments to `printf` are wrapped with type information

Pointer dereferences are checked either statically (optimized) or dynamically (typical)
Another Example:

typedef struct Point { int x, y; } pt;

void addTo(pt *p, pt *q) {
    p->x += q->x;
    p->y += q->y;
}

void foo() {
    pt a = {1, 2};
    pt b = {3, 4};
    pt *aptr = &a;
    pt *bptr = &b;
    addTo(aptr, bptr);
}

Many times, C code such as this compiles directly with no changes needed by programmer.
However, there may be additional run-time checks.

A Better Port

typedef struct Point { int x, y; } pt;

void addTo(pt *p, pt *q) {
    p->x += q->x;
    p->y += q->y;
}

void foo() {
    pt a = {1, 2};
    pt b = {3, 4};
    pt *aptr = &a;
    pt *bptr = &b;
    addTo(aptr, bptr);
}

By refining the types of variables, programmers can often get rid of the overhead.

Making Libraries Robust

struct FILE {
    extern FILE *fopen(char *name, char *mode);
    extern int putc(char, FILE *);
}

void foo() {
    FILE *f = fopen("/tmp/bar.txt", "wb");
    char s[] = "hello";
    int i;
    for (i = 0; i < 5; i++) { putc(s[i], f); }
}

Type error here because f has type FILE* but putc demands FILE*.

Most implementations core dump when given NULL.

One way to fix:

struct FILE {
    extern FILE *fopen(char *name, char *mode);
    extern int putc(char, FILE *);
}

void foo() {
    FILE *f = fopen("/tmp/bar.txt", "wb");
    char s[] = "hello";
    int i;
    for (i = 0; i < 5; i++) { putc(s[i], f); }
}

Dynamically checks that f is an actual file.

A better fix:

struct FILE {
    extern FILE *fopen(char *name, char *mode);
    extern int putc(char, FILE *);
}

void foo() {
    FILE *fn = fopen("/tmp/bar.txt", "wb");
    char s[] = "hello";
    int i;
    if (fn != NULL) {
        for (i = 0; i < 5; i++) { putc(s[i], fn); }
    } else {
        throw new FileError("can't open /tmp/bar.txt!");
    }
}

Object Lifetimes: Spot the Bug

t *add(pt *p, pt *q) {
    pt c;
    c->x = p->x + q->x;
    c->y = p->y + q->y;
    return &c;
}

void foo() {
    pt a = {1, 2};
    pt b = {3, 4};
    pt *c = addTo(&a, &b);
    c->x = 10;
}

x's lifetime ends here!

So dereferencing c here can cause problems.
Tracking Object Lifetimes

- Cyclone uses a region-based type system:
  - Each lexical block is treated as a distinct region.
  - Each pointer type has an associated region: `int* r`
  - The heap is treated as a special region (`H`) with a global lifetime (more on this later).
  - A pointer can only be dereferenced while the region is still live.

Simple Region Example

```c
pt a = {1,2};
void foo() {
  pt b = {3,4};
  pt @ H aptr = &a;
  pt @ foo bptr = &b;
  addTo(aptr, bptr);
}
```

Region Polymorphism

```c
void addTo<r1, r2>(pt * r1 p, pt * r2 q) {
p->x += q->x;
p->y += q->y;
}
```

This is standard parametric polymorphism:

```
addTo: ∀ r1. ∀ r2. (pt * r1 × pt * r2) → void
```

So this would go through...

```c
pt @ H add s1, s2(pt * s1 p, pt * s2 q) {
p @ s = malloc(sizeof(pt));
r->x = p->x + q->x;
r->y = p->y + q->y;
return r;
}
```

```c
pt a = {1,2};
void foo() {
  pt b = {3,4};
  pt @ H c = add@H(s1, s2);
  c->x = 10;
}
```

And this would be caught

```c
pt @ H add s1, s2(pt * s1 p, pt * s2 q) {
p @ r;
r->x = p->x + q->x;
r->y = p->y + q->y;
return &r;
}
```

```c
pt a = {1,2};
void foo() {
  pt b = {3,4};
  pt @ H c = add@H(s1, s2);
  c->x = 10;
}
```
On the other hand...

```c
pt * add(pt * r1, pt * r2) {  
p->x += q->x;  
p->y += q->y;  
return p;  }
pt a = (1,2);  
void foo() {  
    pt b = (3,4);  
    pt * c = add(&a, &b);  
    c->x = 10;  }
```

So we must be explicit

```c
pt * add(pt * r1, pt * r2) {  
p->x += q->x;  
p->y += q->y;  
return p;  }
pt a = (1,2);  
void foo() {  
    pt b = (3,4);  
    pt * c = add(&a, &b);  
c->x = 10;  }
```

What has to be written is thus:

```c
pt * add(pt * r1, pt * r2) {  
p->x += q->x;  
p->y += q->y;  
return p;  }
pt a = (1,2);  
void foo() {  
    pt b = (3,4);  
    pt * c = add(&a, &b);  
c->x = 10;  }
```

Three versions of add:

```c
pt * add(pt * r1, pt * r2) {  
p->x += q->x;  
p->y += q->y;  
return r;  }
pt * r1 add(pt * r1, pt * r2) {  
p->x += q->x;  
p->y += q->y;  
return p;  }
pt * r2 add(pt * r1, pt * r2) {  
q->x += p->x;  
q->y += p->y;  
return q;  }
```

What about the heap?

- Two mechanisms:
  - GC-based:
    - ignore free()
    - let conservative collector reclaim objects.
  - Region-based:
    - allow creation of dynamically growable regions
    - objects are allocated within the region
    - type system tracks region of objects
    - when region is deallocated, all objects are freed
    - type checker ensures you don’t access the region after it’s deallocated.

Growable Regions

```c
typedef struct List<`r> {
    int head;  
    struct List<`r> *`r tl;  
} *`r list_t<`r>;
void foo(unsigned int i) {  
    region<`d> h {
        list_t<`d> x = NULL, temp;  
        for (; i != 0; i++) {  
            temp = rmalloc(h, sizeof(struct List));  
            temp->head = i; temp->tl = x;  
            x = temp;  
        }  
        process(h, x);  
    }  
}
```
Growable Regions

```c
typedef struct List<
    int head;
    struct List<
        * tl;
} * list_t<

void foo(unsigned int i) {
    region<
        list_t<
            x = NULL, temp;
    for (; i != 0; i++) {
        temp = rmalloc(h,
            sizeof(struct List));
        temp->head = i; temp->tl = x;
        x = temp;
    }
    process(h,x);
}
```

The list structure is parameterized by a region.

```
Growable Regions

typedef struct List<
    int head;
    struct List<
        * tl;
} * list_t<

void foo(unsigned int i) {
    region<d> h {
        list_t<d> x = NULL, temp;
        for (; i != 0; i++) {
            temp = rmalloc(h,
                sizeof(struct List));
            temp->head = i; temp->tl = x;
            x = temp;
        }
        process(h,x);
    }
```

The "region<d> h{...}" introduces a new dynamic region 'd' with handle h.

```
Growable Regions

typedef struct List<
    int head;
    struct List<
        * tl;
} * list_t<

void foo(unsigned int i) {
    region<d> h {
        list_t<d> x = NULL, temp;
        for (; i != 0; i++) {
            temp = rmalloc(h,
                sizeof(struct List));
            temp->head = i; temp->tl = x;
            x = temp;
        }
        process(h,x);
    }
```

Since this function is given the handle for 'd', it can allocate objects in there too, or return results in that region.

```
Growable Regions

typedef struct List<
    int head;
    struct List<
        * tl;
} * list_t<

void foo(unsigned int i) {
    region<d> h {
        list_t<d> x = NULL, temp;
        for (; i != 0; i++) {
            temp = rmalloc(h,
                sizeof(struct List));
            temp->head = i; temp->tl = x;
            x = temp;
        }
        process(h,x);
    }
```

The entire region is deallocated at the end of its scope.

```
Growable Regions

typedef struct List<
    int head;
    struct List<
        * tl;
} * list_t<

void foo(unsigned int i) {
    region<d> h {
        list_t<d> x = NULL, temp;
        for (; i != 0; i++) {
            temp = rmalloc(h,
                sizeof(struct List));
            temp->head = i; temp->tl = x;
            x = temp;
        }
        process(h,x);
    }
```

Note that objects within region 'd' can point to objects in any other region.
So Far: Advantages

- Type system makes sure that:
  - can’t dereference a pointer to a freed object.
  - can’t forget to free a region.
  - supports some dangling pointers.
- Runtime system ensures that:
  - region and object allocation are constant time.
  - region deallocation is constant time and faster than individually free’ing the objects.
- So the approach is quite attractive for real-time systems when compared to GC.

Disadvantages

- Programmers have to make the lifetimes of objects explicit:
  - must put regions on return types
  - must allocate objects within regions
  - must pass region handles around.
- Regions lifetimes must be LIFO
  - No easy way to handle objects with overlapping, non-nested lifetimes
- Constructing a region expensive if only allocates a small number of objects

Overcoming the Limitations

- Allow greater control over lifetimes
  - Object lifetimes
    - Unique pointers and reference-counted pointers
  - Region lifetimes
    - Dynamic regions
- But not for nothing …
  - Restrictions on aliasing
  - Possibility of memory leaks
  - Sometimes need (infrequent) runtime checks

Unique Region

- Distinguished region name \( U \)
- Individual objects can be freed manually
- An intraprocedural, flow-sensitive analysis
  - ensures that a unique pointer is not used after it is consumed (i.e. freed)
  - treats copies as destructive; i.e. only one usable copy of a pointer to the same memory
  - Loosely based on affine type systems

Simple Example

```c
void foo() {
    int * U x = malloc(sizeof(int));
    int * U y = x; // consumes x
    *x = 5; // disallowed
    free(y); // consumes y
    *y = 7; // disallowed
}
```

Unique-path access restriction

- Invariant: Only ever one (unique) path by which to access a unique object at any program point

```c
void f(int * U * x, int * U * y) {
    free(*x);
    **y = 1; // error if *x == *y
}
```
Sharing Unique Pointers

- Unique paths can be tracked easily
- But how do we deal with unique pointers accessible via non-unique path? Options:
  - Don’t allow them; norm in linear type systems
  - Track sharable containers
    - E.g., guarded types in Vault; restricts container aliasing, and may not be thread-safe
  - Swap out unique pointers

Swap to the rescue

```c
int *U g = NULL; // can’t access directly
void init(int v) {
  int * U x = malloc(sizeof(int));
  *x = v;
  g :=: x; // atomically swap *x with *g
  free(x);
}
```

Other Manual Mechanisms

- Regions with dynamic lifetime
- Reference-counted pointers
  - Based on unique pointers, to track aliasing.
- Upshot: can really improve memory footprint, and sometimes application throughput.

Performance

- Typically 1.5x C; up to 4-5x
- Bottlenecks
  - Array-bounds checks
  - Unoptimized libraries (e.g. string, file I/O libraries)

Cyclone: where we stand

- Cyclone compiler
  - ~100KL of Cyclone code
  - Bulk is the type-checker and dataflow analyses
  - Straightforward translation to C
  - Available for many architectures (Linux, BSD, Irix, Cygwin, Sparc, etc.)
- Ports
  - Libc and other libs (sockets, XML, lists, and more)
  - bison, flex, web server, cfrac, grobner, NT device driver ... (~40KL total)
  - Typically differ from original C by 5-15%

Tools and Applications

- Lex, Bison, Memory profiler
- Semi-automated porting tools
  - Guess whether to convert a C * to Cyclone *, @, or ?
- In-kernel transport protocols (SOSP 03)
- Streaming data overlay networks (OPENARCH 03)
- In-kernel extensions (OPENARCH 02)
- Hardware description languages
**Summary**

- Research in safe, low-level languages is crucial.
- Programmer-controlled data representations and memory management are critical issues.
- We have good typing technologies at this point, but adapting them to practical settings is a lot of work.
- Cyclone isn’t a full solution but it’s moving in the right direction.

**Obligatory URL**

http://www.cs.umd.edu/projects/cyclone

- Includes code, papers, documentation, and more!