Cyclone
A Safe Dialect of C
Michael Hicks
University of Maryland, College Park

Credit where credit is due ...

• Cyclone is a research language, the product of many collaborators:
  - Greg Morrisett, Yanling Wang (Harvard)
  - Dan Grossman (Washington)
  - Nikhil Swamy (Maryland)
  - Trevor Jim (AT&T)

1988? 2005?

• “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.”

• More than 15 years later, nearly half of CERT advisories involve buffer overruns, format string attacks, and similar low-level attacks.

The C Programming Language

• Critical software is often coded in C
  - device drivers, kernels
  - file systems, web servers, email systems
  - switches, routers, firewalls

• … most arguably because it is low-level
  - Control over data structure representations
  - Control over memory management
  - Manifest cost: good performance

Low-level, but unsafe

• 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.)

What about Java?

• Java provides safety in part via:
  - hidden data fields and run-time checks
  - automatic memory management

• Data representation and resource management are essential aspects of low-level systems
Some possible approaches

- Compile C with extra information
  - type fields, size fields, live-pointer table, ...
  - treats C as a higher-level language
- Use static analysis
  - very difficult, not perfect
  - less modular
- Ban unsafe features
  - there are many
  - you need them

Cyclone

A safe, convenient, and modern language at the C level of abstraction

- Safe: memory safety, abstract types; fail-stop
- C-level: user-controlled data representation and resource management, easy interoperability, “manifest cost”
- Convenient: may need more type annotations, but work hard to avoid it
- Modern: add features to capture common idioms

“New code for legacy or inherently low-level systems”

Outline

Status

- >110K lines of Cyclone code
  - 80K compiler, libraries
  - 30K various servers, applications, device drivers
- gcc back-end (Linux, Cygwin, OSX, LEGO, …)
- User’s manual, mailing lists, …
- Still a research vehicle

Projects using Cyclone

- Open Kernel Environment [Bos/Samwel, OPENARCH 02]
- MediaNet [Hicks et al, OPENARCH 03]
- RBClick [Patel/Lepreau, OPENARCH 03]
- STP [Patel et al., SOSP 03]
- FPGA synthesis [Teifel/Manohar, ISACS 04]
- O/S class at Maryland [2004-2005]

Status

- >110K lines of Cyclone code
  - 80K compiler, libraries
  - 30K various servers, applications, device drivers
- gcc back-end (Linux, Cygwin, OSX, LEGO, …)
- User’s manual, mailing lists, …
- Still a research vehicle

What is a C buffer overflow?

```c
#include <stdio>

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

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

User types login

101st character ...

Stack smashed!

Abstraction-violating Attack

Two kinds of Pointer Errors

- Language and library abstractions may not be enforced
  - Array accesses, pointer dereferences, type casts, format strings “trusted” by the compiler
- Other attacks exploit this fact
  - Heap-based buffer overruns
  - Format string attacks

- Spatial
  - Dereferencing outside of a legal memory buffer, possibly at the wrong type
  - Abstraction-violating attacks in this category
- Temporal
  - Dereferencing a pointer after the pointed-to buffer has been freed
Preventing Spatial Errors

- Don’t allow dereferencing a pointer unless compiler can prove it’s safe
  - Often too conservative
- Prevent dereferencing with *dynamic* checks
  - May be able to eliminate some or all of these with static analysis, or programmer-provided type annotations
  - Safety first; then tune performance

Thin Pointers

- A “thin” pointer (one word)

```
*pp = NULL;
*p
```

Only dereference permitted, no bounds check needed

```
p = NULL;
*p
```

May be null

```
p = NULL;
*p
```

Requires a null check if so

```
p = NULL;
*p
```

Types and qualifiers for more flexibility and/or fewer checks

```
char *p;
char * @notnull p1; // illegal: p1=NULL;
p1 = p
char * @notnull @numelts(6) p2; // illegal: p2=p1
```
**Thin Pointers**

Shorthand

```c
char *p;
char @ p1;
char @{6} p2;
```

**Fat Pointers**

A “fat” pointer; has run time bounds: 3-word representation

```c
q++; q[0] q--; q--; q[0]
```

**Fat Pointers**

Pointer arithmetic OK

```c
q++; q[0] q--; q--; q[0]
```

**Fat Pointers**

Bounds check on dereference

```c
q++; q[0] q--; q--; q[0]
```

**Fat Pointers**

Dangling pointers OK...

```c
q++; q[0] q--; q--; q[0]
```
Fat Pointers

H E L L O

q:

q++; q[0]
q--; q--; q[0]

… caught on dereference

Types and qualifiers
char * @fat q;

Shorthand
char ? q;

Thin Pointer, Dynamic Bounds

p

len = 5

void foo(int len, char * @numelts(len) p) {
    for (int i = 0; i<len; i++)
        p[i] = …
}

Our buildlib tool easily generates platform dependent headers with signatures like these (programmer helps)

Pointer Qualifier Summary

- @fat
  - rep. as a triple: [base, upper, curr]
  - supports all C pointer ops
  - but any dereference (may be) checked
- @nonnull
  - Obviates null check (compiler must prove)
- @numelts(n)
  - Obviates bounds check (compiler must prove)
  - Can refer to dynamic lengths
- @zeroterm
  - Pointer is zero-terminated

Interfacing with libC

FILE *fopen(char * @notnull @zeroterm name, char * @notnull @zeroterm mode);

int putc(char, FILE * @notnull);

… fgets(char * @zeroterm @numelts(len), int len, FILE * @notnull);
Temporal Errors

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

r's lifetime ends here!
so dereferencing c here can cause problems...

Preventing Temporal Errors

- Tracks object lifetimes by associating a region with each pointer:
  ```c
  int* @region(r)
  - A pointer can only be dereferenced while the region is still live. int *r for short.
  - Two basic kinds of regions
    - A lexical block (i.e., an activation record)
    - The heap (`H); has a global lifetime.

Simple Region Example

```c
pt a = {1,2};
void foo() {
    pt b = {3,4};
    pt @H aptr = &a;
    pt @foo bptr = &b;
    addTo(&a, &b);
}
```
a lives in the heap region, so &a has type pt @H.
b lives in the activation record of foo so &b has type pt @foo.
region inference can figure out the regions, so the programmer doesn't have to write them

So this would go through...

```c
pt *H add<`r1,`r2>(pt *`r1 p, pt *`r2 q) {
    pt r = malloc(sizeof(pt));
    r->x = p->x + q->x;
    r->y = p->y + q->y;
    return r;
}
void foo() {
    pt b = {3,4};
    pt *H c = add<`H,`foo>({a, &b});
    pt *H d = add<`foo,`foo>({b, &b});
    c->x = 10;
}
```

Region Polymorphism

```c
void addTo<`r1,`r2>(pt *`r1 p, pt *`r2 q) {
    p->x += q->x;
    p->y += q->y;
}
```
addTo is parameterized by the regions for p and q.
This is standard parametric polymorphism: addTo: \( \forall \`r1. \forall \`r2. (pt *\`r1 \times pt *\`r2) \rightarrow void \)

And this would be caught

```c
pt *H add<`r1,`r2>(pt *`r1 p, pt *`r2 q) {
    pt r;
    r->x = p->x + q->x;
    r->y = p->y + q->y;
    return &r;
}
void foo() {
    pt b = {3,4};
    pt *H c = add<`H,`foo>({a, &b});
    pt *H d = add<`foo,`foo>({b, &b});
    c->x = 10;
}
```
On the other hand...

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

region of p is `r1, not `H

So we must be explicit

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

What has to be written is thus:

```c
pt * r1 add(pt * p, pt * q) {
    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);
    pt * d = add(&b, &b);
    c->x = 10;
}
```

The types say it all

```c
pt * add (pt * p, pt * q);
pt * r1 add (pt * r1 p, pt * q);
pt * r2 add (pt * p, pt * r2 q);
char * @zeroterm `r fgets(char *{len} @zeroterm `r, int len, FILE @);
```

Dynamic Allocation

- How can we be sure data is live?
- Can use a GC, or safe manual techniques
- GC-based
  - Data allocated in heap is given region `H
    - Region is always live; all dereferences are always safe
    - Conservative collector reclaim dead objects
  - Simple, but little control over performance
    - Potentially significant memory overheads
    - Pause times
    - May not be feasible in some environments

Safe Manual Techniques

- Approach: generalize regions, track pointers
- New region kind: Arenas
  - Dynamic allocation, but all objects freed at once
- And/or impose aliasing restrictions
  - Can free individual objects using malloc/free or reference counting if aliasing is tracked
- When writing apps, use GC first, tune as necessary
  - Can result in significantly improved memory footprint and throughput
LIFO Arenas

- Dynamic allocation mechanism
- Lifetime of entire arena is scoped
  - At conclusion of scope, all data allocated in the arena is freed
  - Like a stack frame, but permits dynamic allocation
    - Useful when caller doesn’t know how much memory needed by callee, but controls lifetime

LIFO Arena Example

```c
FILE *infile = ...
Image *i;
if (tag(infile) == HUFFMAN) {
    region<`r> h;  // region `r created
    struct hnode * `r huff_tree;
    huff_tree = read_tree(h, infile);
    // dynamically allocates with h
    i = decode_image(infile, huff_tree,...);
    // region `r deallocated upon exit of scope
} else ...
```

Unique Pointers

- If object is known to have no aliases, it can be freed manually
  - Pointer qualifier @aqual(\U), or \U for short
- An intraprocedural, flow-sensitive analysis
  - ensures that a unique pointer is not used after it is consumed (i.e. freed)
  - treats copies as destructive
    - one usable copy of a pointer to the same memory

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
}
```

Temporary Aliasing

- Problem: Non-aliasing too restrictive
- Partial solution: Allow temporary, lexically-scoped aliasing under acceptable conditions
  - Makes tracked pointers easier to use
  - Increases code reuse

Alias Construct

```c
extern void f(int * `r x);  // `r any region

void foo() {
    int *\U x = malloc(sizeof(int));
    *x = 3;
    { alias < `> int * `r y = x;  // `r fresh
        f(y);                      // y aliasable, but x consumed
    }
    free(x);
}
```
With inference

```c
extern void f(int * x);

void foo() {
    int *x = malloc(sizeof(int));
    *x = 3;
    f(x); // alias inserted here automatically
    free(x);
}
```

Reference-counted Pointers

- **Aliasing qualifier** `\RC`
  - pointed-to data have hidden count field
- **Aliasing tracked as with unique pointers. Explicit aliasing/freeing via**
  ```c
  `a *\RC`r alias_refptr(`a *\RC`r);
  void drop_refptr(`a *\RC`r);
  ```

Interesting Combinations

- Tracked pointers can be freed manually, with `free` or `drop_refptr`, or automatically
  - Pointers into the heap freed by GC
  - Pointers into LIFO arenas freed at end of scope
    - Called a **reap** by Berger et al
  - Can use tracked pointers to **keys** to permit arenas to have non-lexical lifetimes
    - Lifetime of arena corresponds with the liveness of the key
    - Called **dynamic arena**

Summary

<table>
<thead>
<tr>
<th>Region Variety</th>
<th>Allocation (objects)</th>
<th>Deallocation (what)</th>
<th>Deallocation (when)</th>
<th>Aliasing (objs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>static</td>
<td>whole region</td>
<td>exit of scope</td>
<td>manual</td>
</tr>
<tr>
<td>LIFO</td>
<td>dynamic</td>
<td>single objects</td>
<td>manual</td>
<td>GC</td>
</tr>
<tr>
<td>Dynamic</td>
<td></td>
<td></td>
<td></td>
<td>manual</td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
<td>restricted</td>
</tr>
<tr>
<td>Unique</td>
<td></td>
<td></td>
<td></td>
<td>manual</td>
</tr>
<tr>
<td>Refcounted</td>
<td></td>
<td></td>
<td></td>
<td>manual</td>
</tr>
</tbody>
</table>

Ensuring Uniformity and Reuse

- Many different idioms could be hard to use
  - Duplicated library functions
  - Hard-to-change application code
- We have solved this problem by
  - Using region types as a unifying theme
  - Region (and aliasing) polymorphism
    - E.g., functions independent of arguments’ regions/aliasing
  - All regions can be treated as if lexical
    - Temporarily, under correct circumstances
    - Using alias and open (for dynamic arenas)

Some Application Experience

- **Boa** web server
- **Cfrac** Prime factorization
- **BetaFTPD** ftp server
- **Epic** image compression
- **Kiss-FFT** portable fourier transform
- **MediaNet** streaming overlay network
- **Linux Drivers** net, video, sound
- **CycWeb** web server
- **CycScheme** scheme interpreter
Application Characteristics

<table>
<thead>
<tr>
<th>Program</th>
<th>Non-comment Lines of Code</th>
<th>Manual Mechanisms</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
<td>C (+manual)</td>
</tr>
<tr>
<td>Boa</td>
<td>5217 ± 286 (5%)</td>
<td>± 98 (1%)</td>
</tr>
<tr>
<td>Cfarc</td>
<td>3143 ± 183 (5%)</td>
<td>± 784 (25%)</td>
</tr>
<tr>
<td>BetaftpD</td>
<td>1164 ± 219 (18%)</td>
<td>± 238 (22%)</td>
</tr>
<tr>
<td>Epic</td>
<td>2123 ± 218 (10%)</td>
<td>± 117 (6%)</td>
</tr>
<tr>
<td>KissFFT</td>
<td>453 ± 74 (16%)</td>
<td>± 25 (5%)</td>
</tr>
<tr>
<td>S13Btoo</td>
<td>1972 ± 971 (40%)</td>
<td>± 312 (14%)</td>
</tr>
<tr>
<td>S10audio</td>
<td>2598 ± 1300 (57%)</td>
<td>± 318 (10%)</td>
</tr>
<tr>
<td>pwe</td>
<td>3755 ± 1373 (36%)</td>
<td>± 1024 (26%)</td>
</tr>
<tr>
<td>MediaNet</td>
<td>8715 ± 320 (4%)</td>
<td></td>
</tr>
<tr>
<td>CycWeb</td>
<td>667</td>
<td>U</td>
</tr>
<tr>
<td>CycScheme</td>
<td>2523</td>
<td></td>
</tr>
</tbody>
</table>

U = unique pointers  R = ref-counted pointers  L = LIFO regions  D = dynamic arenas

Experimental Measurements

- **Platform**
  - Dual 1.6 GHz AMD Athlon MP 2000
  - 1 GB RAM
  - Switched Myrinet
  - Linux 2.4.20 (RedHat)
- **Software**
  - C code: gcc 3.2.2
  - Cyclone code: cyclone 0.9
  - GC: BDW conservative collector 6.2α4
  - malloc/free: Lea allocator 2.7.2

Bottom Line

- **CPU time**
  - I/O bound applications have comparable performance
  - All applications: at most 60% slowdown
  - GC has little impact on elapsed time
  - MediaNet is the exception
- **Memory usage**
  - Using GC requires far more memory than manual
  - Cyclone manual techniques approach footprint of C original

Throughput (Webservers)

Throughput (MediaNet)

Memory Usage (Web)
Things I didn’t talk about

- Modern language features too
  - Tagged unions and data types
  - Pattern matching
  - Exceptions
  - Allocation with `new`
- Porting tool
- Lots of libraries
Related Work: making C safer

- Compile to make dynamic checks possible
  - Safe-C [Austin et al.], RTC [Yong/Horwitz], ...
  - Purify, Stackguard, Electric Fence, ...
  - CCured [Necula et al.]
    - performance via whole-program analysis
    - less user burden
    - less memory management, single-threaded
- Control-C [Adve et al.] weaker guarantee, less burden
- SFI [Wahbe, Small, …]: sandboxing via binary rewriting

Related Work: Checking C

- Model-checking C code (SLAM, BLAST, …)
  - Leverages scalability of MC
  - Assures (weak) memory safety
- Lint-like tools (Splint, Metal, PreFIX, …)
  - Good at reducing false positives
  - Cannot ensure absence of bugs
  - Metal particularly good for user-defined checks
- Cqual (user-defined qualifiers, lots of inference)
  Better for unchangeable code or user-defined checks

Related work: higher and lower

- Adapted/extended ideas:
  - polymorphism [ML, Haskell, …]
  - regions [Tofte/Talpin, Walker et al., …]
  - safety via dataflow [Java, …]
  - existential types [Mitchell/Plotkin, …]
  - controlling data representation [Ada, Modula-3, …]
- Safe lower-level languages [TAL, PCC, …]
  - engineered for machine-generated code
- Vault: stronger properties via restricted aliasing

Future Work

- Tracked pointers sometimes painful; want
  - Better inference (e.g. for alias)
  - Richer API (restrict; autorelease)
- Prevent leaks
  - unique and reference-counted pointers
- Specified aliasing
  - for doubly-linked lists, etc.
- Concurrency

Conclusions

- High degree of control, safely:
- Sound mechanisms for low-level control
  - Checked pointers for spatial errors
  - Variety of techniques for temporal errors
    - Region-based vs. object-based deallocation
    - Manual vs. automatic reclamation
- Region- and alias-annotated pointers within a coherent framework
  - Scoped regions unifying theme (alias,open)
  - Polymorphism, for code reuse

More Information

- Cyclone homepage
  http://www.cs.umd.edu/projects/cyclone/
- Has papers, benchmarks, distribution