Cyclone: A Type-Safe Dialect of C

Dan Grossman Michael Hicks Trevor Jim Greg Morrisett

October 19, 2004

If any bug has achieved celebrity status, it is the buffer overflow. It made front-page news as early as 1987, as the enabler of the Morris worm, the first worm to spread through the Internet. In recent years, attacks exploiting buffer overflows have become more frequent, and more virulent. This year, for example, the Witty worm was released to the wild less than 48 hours after a buffer overflow vulnerability was publicly announced; in 45 minutes, it infected the entire world-wide population of 12,000 machines running the vulnerable programs.

Notably, buffer overflows are a problem only for the C and C++ languages—Java and other "safe" languages have built-in protection against them. Moreover, buffer overflows appear in C programs written by expert programmers who are security concious—programs such as OpenSSH, Kerberos, and the commercial intrusion detection programs that were the target of Witty.

This is bad news for C. If security experts have trouble producing overflow-free C programs, then there is not much hope for ordinary C programmers. On the other hand, programming in Java is no panacea; for certain applications, C has no competition. From a programmer's point of view, all the safe languages are about the same, while C is a very different beast.

Cyclone

Cyclone is an effort to bring safety to C, without turning it into another Java. We can sum up why we prefer C to Java in two words: *transparency* and *control*. A few examples will make this clear.

- In C, an array of structs will be laid out contiguously in memory, which is good for cache locality. In Java, the decision of how to lay out an array of objects is made by the compiler, and probably has indirections.
- C has data types that match hardware data types and operations. Java abstracts from the hardware ("write once, run anywhere").
- C has manual memory management, whereas Java has garbage collection. Garbage collection is safe and convenient, but places little control over performance in the hands of the programmer, and indeed encourages an allocation-intensive style.

In short, C programmers can see the costs of their programs simply by looking at them, and they can easily change data representations and fundamental strategies like memory management. It's easy for a C programmer to tune their code for performance or for resource constraints.

Cyclone is a dialect of C that retains its transparency and control, but adds the benefits of safety (i.e., no unchecked run-time errors.) In Cyclone, buffer overflows and related bugs are prevented for *all* programs, whether written by a security expert or by a novice. The changes required to achieve safety are pervasive, but Cyclone is still recognizably C; in fact, a good way to learn 80% of Cyclone is to pick up Kernighan and Ritchie.

Safety has a price. Figure 1 shows the performance of Cyclone and Java code normalized to the performance of C code for most of the micro-benchmarks in the *Great Programming Language Shootout* (see `http://shootout.alioth.debian.org/`.) Though these micro-benchmarks should be taken with a large

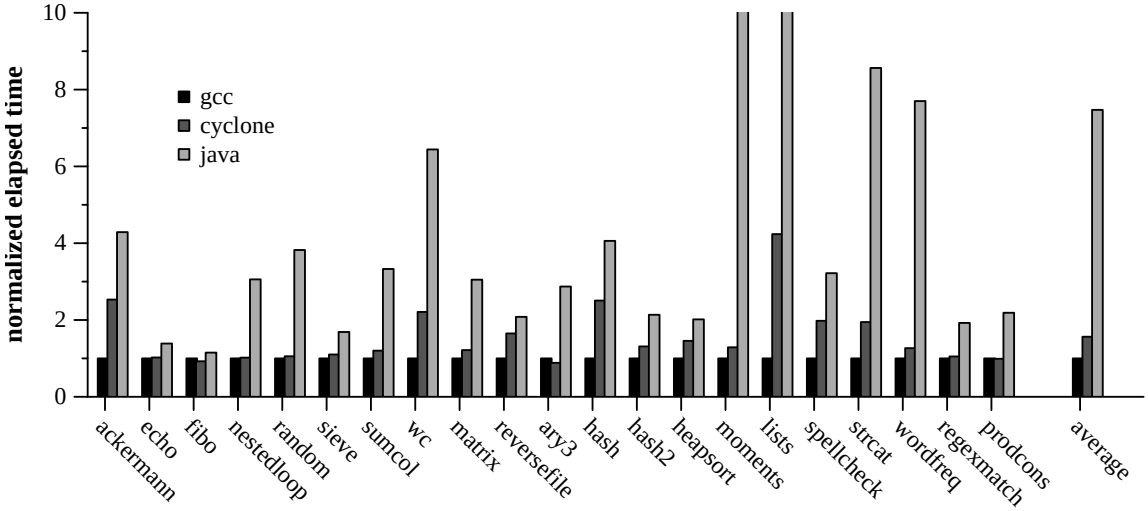

Figure 1: Great Programming Language Shootout Performance for C, Cyclone, and Java

grain of salt, they do give a rough idea of the relative performance of the different languages.

The benchmarks were run on a dual 2.8GHz/2GB Red Hat Enterprise workstation. We used Cyclone version 0.8.2 with the `-O3` flag, Sun's Java client SDK build version 1.4.2_05-b04, and GCC version 3.2.3 with the `-O3` flag. To avoid measuring start-up and compilation time for the Java VM, we measured elapsed time after a warmup run. We also did measurements using Sun's server SDK and the GNU Java compiler `gcj`, and the results were essentially the same as the Sun client SDK results. Each reported number in Figure 1 is the median of 11 trials; there was little variance.

The average over all of the benchmarks (plotted on the far right) shows that Cyclone is about 0.6 times slower than GCC, whereas Sun's Java is about 6.5 times slower. However, the *moments* and *lists* benchmarks are outliers for which the Java code has very bad performance (15× and 74× slowdown respectively—so bad that we've had to clip the graph.) If we assume these were poorly coded and remove them from the averages, then we still see a factor of 3 slowdown for the Java code, compared to 0.4 for Cyclone.

In addition to time, programmers may be worried about space. We have found that, again, there are overheads when using Cyclone as compared to C, but these overheads are much less than for Java. For instance, the C version of the *heapsort* benchmark had a maximum resident working set size of 472 4KB pages, the Cyclone version used 504 pages, and the Java version used 2,471.

There is another cost to achieving safety, namely, the cost of porting a program from C to a safe language. Porting a program to Java essentially involves a complete rewrite, for anything but the simplest programs. In contrast, most of the Shootout benchmarks can be ported to Cyclone by touching 5 to 15 percent of the lines of code. To achieve the best performance, programmers may have to provide additional information in the form of *extended type qualifiers*, which we describe below. Of course, the number of lines of code that changed tells us little about how hard it is to make those changes. In all honesty, this can still be a time-consuming and frustrating task. Nevertheless, it is considerably easier than rewriting the program from scratch in a new language.

Like Java, Cyclone gives strong safety guarantees. There are some overheads, both in terms of run-time performance and porting costs, but the overheads of

Cyclone compare quite favourably to Java. For the rest of the article, we will describe some features of Cyclone that allow us to achieve safety while minimizing these costs.

Pointers

A common way of preventing buffer overruns in languages such as Java is to use dynamic bounds checks. For example, in Java whenever you have an expression like `arr[i]`, the VM may check at run time whether `i` is within the bounds of the array `arr`. While this check may get optimized away, the programmer has no way to be sure of this. Moreover, the programmer has no control over the memory representation of the array, e.g., to interact with hardware-defined data structures in an operating system. Finally, Java does not allow pointer arithmetic for iterating over the array, which is quite common in C code.

Cyclone provides a middle ground between C and Java when it comes to pointers. On the one hand, Cyclone will perform dynamic checks when it cannot be sure that dereferences are in bounds, and throw an exception when necessary. On the other hand, the programmer can choose from a variety of *pointer qualifiers* to have control over when such checks may occur, and how pointers are represented in memory. Cyclone basically supports three kinds of pointers: *fat* pointers, *thin* pointers, and *bounded* pointers.

Fat Pointers

A *fat* pointer is similar to a Java array, in that it might incur a dynamic bounds check. A fat pointer is denoted by writing the qualifier `@fat` after the `*`. For example, Figure 2 shows a program that echoes its command-line arguments. Except for the declaration of `argv`, which holds the command-line arguments, the program looks just like you would write it in C: pointer arithmetic (`argv++`) is used to move `argv` to point to each argument in turn, so it can be printed. The difference is that a `@fat` pointer comes with bounds information and is thus "fatter" than a traditional pointer. Each time a fat pointer is dereferenced or its contents are assigned to, Cyclone inserts a bounds check. This guarantees that a `@fat` pointer can never cause a buffer overflow.

```
#include <stdio.h>
int main(int argc, char *@fat *@fat argv) {
  argc--; argv++; /* skip command name */
  while (argc > 0) {
    printf(" %s",*argv);
    argc--; argv++;
  }
  printf("\n");
  return 0;
}
```

Figure 2: Echoing command-line arguments

Because of the bounds information contained in `@fat` pointers, `argc` is superfluous: you can get the size of `argv` by writing `numelts(argv)`. We've kept `argc` as an argument of `main` for backwards compatibility. Because `@fat` pointers are common, one can abbreviate `* @fat` as `?` (question mark). So we could write `char *@fat *@fat` as simply `char ??`. Quite often, porting C code to Cyclone merely means changing some `*`'s to `?`'s.

Thin Pointers

Many times, there is no need to include bounds information on a pointer. For example, if we declare

```
int x = 3;
int *y = &x;
```

then it is clear that `y` is a pointer to a *single* integer 3 (the contents of `x`). Just as in C, `y` is represented by a memory address (namely, the address of `x`); this is why we call it a *thin* pointer. A dereference of a thin pointer (e.g., with syntax `*y`) will never incur a bounds check. To ensure that a dereference is always in bounds, you can't do pointer arithmetic on a `*` pointer.

Bounded Pointers

You can also define thin pointers to buffers containing more than one element using the `@numelts` qualifier; we call these *bounded pointers*. For instance, we can write:

```
int x[4] = {1,2,3,4};
int *@numelts(4) arr = x;
```

Here, the variable `arr` is a pointer to a sequence of four integer values. In fact, the type "`int *`" is just short-hand for "`int *@numelts(1)`". Accessing a bounded pointer like `arr` via the expression `arr[i]` may incur a bounds check if `i` cannot be proven in-bounds by the compiler. However, because the bound is known to the compiler, the representation of the pointer is still a single memory address.

Bounded pointers can also be used to correlate a pointer to an array whose length is *not* known to the compiler with a variable that defines it. For example, C programmers often write code like the following:

```
int sum(int num, int *p) {
  int a = 0;
  for (unsigned i = 0; i < num; i++)
    a += p[i];
}
```

Here, `num` is the length of the array pointed at by `p`. In Cyclone, this relationship can be expressed by giving `sum` the following type (the body of the function is the same):

```
int sum(tag_t num, int p[num])
```

The type of `num` is specified as `tag_t`, which is simply a `const unsigned int` that may appear as an array bound, in this case for `p`. A bounded pointer paired with a `tag_t` is quite similar to a fat pointer. In general, the programmer can convert freely between the different kinds of pointers to balance the needs of representation, performance, and ease-of-use.

Initialization and NULL

In the above discussion, we have assumed that thin and bounded pointers address legal memory locations and that fat pointers have correct bounds information. We ensure this using two techniques: definite initialization and NULL-checking.

```
Form *f;
switch (event->eType) {
case frmOpenEvent:
  f = FrmGetActiveForm(); ...
case ctlSelectEvent:
  i = FrmGetObjectIndex(f, field); ...
}
```

Figure 3: Incorrect initialization in C

Definite initialization is a guarantee that a pointer variable will not be dereferenced before it is initialized; this is ensured by source code analysis. For example, the compiler will flag the code in Figure 3 as illegal: the second `case` possibly uses `f` before it is initialized. Java (and indeed many C compilers) include a similar analysis to check that variables are initialized before they are used, but the analysis does not extend to members of objects. Rather, in the case of Java, the VM automatically initializes members with a default value (e.g., NULL). In contrast, Cyclone's analysis extends to struct, union members, and pointer contents to ensure everything is initialized before it is used. This has two benefits: First, we tend to catch more bugs this way, and second, programmers don't pay for the overhead of automatic initialization on top of their own initialization code.

A pointer variable may be NULL, and if it's dereferenced, this could result in a crash. Again, like Java, Cyclone inserts a NULL check to ensure that a pointer can be safely dereferenced. However, the programmer can prevent such a check in two ways. One way is to perform a manual NULL check, as in

```
if (y != NULL) {
  *y = 1; // no check
  *y = 2; // no check
}
```

The second way is to prevent a pointer from ever having NULL as a legal value, using Cyclone's `@notnull` qualifier. For example, consider the `getc` function:

```
int getc(FILE *fp);
```

Most implementations of `getc` assume that `fp` will not be NULL, so if you call `getc(NULL)` you are likely to get a segmentation fault. To prevent this, we can declare

```
int getc(FILE *@notnull fp);
```

indicating that `getc` expects a non-NULL `FILE` pointer as its argument. This simple change tells Cyclone that it does not need to insert NULL checks into the *body* of `getc`. If `getc` is called with a possibly-NULL pointer, Cyclone will insert a NULL check at the *call*:

```
extern FILE *f;
getc(f);            // NULL check here
```

Never-NULL pointers are a perfect example of Cyclone's design philosophy: safety is guaranteed, automatically if possible, and the programmer has control over where any needed checks are performed. The same philosophy applies to initialization: safety is guaranteed, but programmers retain control over how the initialization is accomplished.

Unions and Tagged Unions

To ensure safety, Cylone must prevent the programmer from treating an arbitrary value as if it were a pointer. Thus, the language rules out code such as:

```
int i = 0xbad;
int *p = (int *)i;
*p = 42;
```

But there are other ways in C to convert integers to pointers. In particular, we can use a union to accomplish this:

```
union U { int i; int *p; };

union U x;
x.i = 0xbad;
*x.p = 42;
```

In fact, the C standard says that if you read out any member of a union other than the last one written, the result is undefined. To ensure safety, Cyclone makes a stronger requirement that you can't read out a value from an ordinary union that might contain pointers. However, the language does allow code such as:

```
int j = 42;

int jaddress() {
  union U x;
  x.p = &j;
  return x.i;
}
```

because this converts a pointer to an integer.

Unfortunately, this restriction rules out a lot of C code that uses unions. However, most well-structured code uses some extra information to record what type of value was last written into the union. For instance, it's not uncommon to see definitions similar to:

```
enum tag { Int, Ptr };
union U { int i; int *p; };
struct S { enum tag t; union U u; };

void pr(struct S x) {
  if (x.tag == Int)
    printf("int(%d)",x.u.i);
  else
    printf("ptr(%d)",*x.u.p);
}
```

Here, struct `S` values include a tag indicating which member was last written into the union. A consumer of `S` values, such as the `pr` function, checks the tag to see which member was last written. The problem is that, of course, nothing in C prevents us from having a tag and union that are out of sync.

To avoid this problem, Cyclone provides a built-in form of tagged union and always ensures that the tag is correlated with the last member written in the union. In particular, whenever a tagged union member is updated, the compiler inserts code to update the tag associated with the union. Whenever a member is read, the tag is consulted to ensure that the

member was the last one written. If not, an exception is thrown.

Thus, the example above can be re-written in Cyclone as follows:

```
@tagged union U { int i; int *p; };

void pr(union U x) {
  if (tagcheck(x.i))
    printf("int(%d)",x.i);
  else
    printf("ptr(%d)",*x.p);
}
```

The `@tagged` qualifier indicates to the compiler that `U` should be a tagged union. The operation `tagcheck(x.i)` returns true when `i` was the last member written so it can be used to extract the value. (Alternatively, one can use an extension of `switch` that supports pattern matching to test a large number of tags at once.)

Why did we provide both tagged and untagged unions? In part, we didn't want to force the overhead of tags onto programmers when they weren't necessary to support safety. Additionally, we wanted to make it clear that, like fat pointers, tagged unions include additional run-time information.

Cyclone also provides other run-time type mechanisms that can be used to support functions such as `printf` that need to test the type of some value. In essence, when you call `printf`, the compiler automatically constructs a fat pointer to stack-allocated, tagged values representing the list of arguments. These facilities ensure that vararg functions such as `printf` don't end up "trusting" the format string to accurately describe the arguments. Instead, we can dynamically compare the format string against the run-time type information and throw an exception if there is a mismatch. This ensures that certain format string attacks cannot be used to take over a machine.

We inject run-time type tags of this sort only when the callee demands it for safety. In all other cases, we use the same data representation as C.

Memory Management

In most type-safe languages, programmers do not have direct control over memory management. For instance, a Java programmer cannot force an array to be stack-allocated, nor can she choose to deallocate a heap object with a call to `free`. Instead, all the memory management decisions are handled by the compiler and run-time system (i.e., the garbage collector). This avoids a class of nasty problems that can break type safety. Consider this contrived example which GCC happily compiles, but has a nasty problem:

```
void g(int **zptr) {
  int a = 0;
  *zptr = &a;
}

void poke(int *z) {
  int *q = NULL;
  *z = 0xbad;
  *q = 42;
}

int main() {
  int i = 0;
  int *z = &i;
  g(&z);
  poke(z);
}
```

In this program, `main` calls `g` passing it a pointer to `z`. Then `g` creates a local variable `a` and assigns a pointer to `a` into `z`. So, after returning from the call to `g`, `z` no longer points to `i` but rather to some stack space that used to hold `a`. At this point, we invoke `poke` passing it `z`.

Notice that `poke` declares a new local variable `q` which is likely to be stored in the stack location where `a` was previously. That is, we could very easily end up with `z == &q`, yet `z` is supposed to be a pointer to an integer, not a pointer to a pointer to an integer! Nonetheless, the C type system has failed to signal an error, so we are able to assign an arbitrary integer value to `*z`, thereby changing the value of `q` to point

to an arbitrary location in memory. Then attempting to write to `*q` results in a core dump.

This example can be easily transformed to allow reading or writing arbitrary values to arbitrary locations, thereby violating type safety. Of course, we could just rule out stack allocation (as in Java) but that would introduce overheads and make porting more difficult.

Instead, we use a *region*-based type system to avoid these problems. Each object in Cyclone lives in a conceptual container called a region. For instance, the variables of a function `f` live in a region corresponding to the function's stack frame.

Internally, the type-checker keeps track of the region for each object, and the region into which a pointer value points. For example, when checking the function `g`, the type-checker knows that `a` lives in `g`'s region and thus `&a` is a pointer into `g`'s region. Indeed, the full type of `&a` is written `int*‘g` reflecting the region of the pointer into the type.

The type system rejects programs that either (a) try to let a pointer escape the scope if its region or (b) try to assign a pointer value to a variable with an incompatible region type. In the example above, Cyclone rejects the program at the assignment of `*zptr = &a`. Informally, the reasoning is that since `&a` is a pointer into `g`, the contents of `zptr`, namely `z`, must be a pointer into region `g`. But since the pointer to `z` was passed in to `g`, `z`'s region type had to be defined *outside* of `g`. Thus, `*z` cannot have a type that mentions `g`! In short, the Cyclone type-checker has discovered that the assignment is inconsistent, signalling a problem with a dangling pointer.

In general, the region type system does an excellent job of allowing programmers to pass pointers to stack-allocated objects into functions, but successfully prevents them from leaking back out either explicitly via return, or implicitly through assignments. However, it is sometimes necessary for programmers to provide explicit region annotations to convince the type-checker that an assignment is okay. For example, consider:

```
void f(int **z, int *a) {
  *z = a;
}
```

The type-checker cannot tell whether the assignment would cause a region violation so it rejects the program. If we instead tell the compiler that `*z` and `a` are meant to point into the *same* region, then the type-checker can validate the code. This can be accomplished as follows:

```
void f(int *‘r* z, int *‘r a) {
  *z = a;
}
```

Here, the types of `a` and `*z` are given as `int *‘r` reflecting that they must be pointers into the same (unknown) region `r`.

Of course, stack allocation is only part of the story. How does Cyclone handle dynamically-allocated storage? There are actually a number of options provided by the language, ranging from a garbage-collected heap region, to Apache-style arena regions, to reference-counted objects and regions, and objects that live in their own unique region. Like the stack-allocated regions, each of these mechanisms has certain restrictions that ensure safety, but they all live in the same conceptual region framework as far as the type-checker is concerned. This makes it possible to write re-usable libraries that are independent of the particular kind of region used to hold a data structure.

For most applications, the combination of stack allocation and garbage-collected heap allocation has proven to be simple and effective. However, the other facilities make it possible to fine-tune the storage management strategy to achieve significant wins in space, throughput, and latency.

Other Features

There are a few other features in Cyclone that make programming in the language easier, some of which are close to features in C++. For instance, the language provides support for exceptions, namespaces, subtyping and *parametric polymorphism* (i.e., generics). The support for subtyping and generics makes it possible to write re-usable data structures (e.g., hash-tables) and algorithms. However, in keeping with

the spirit of C, the Cyclone approach supports separate type-checking and compilation of generic definitions from their uses (which is important for shared libraries that may be dynamically linked). In particular, the implementations of generics do not show up in interfaces, and thus a change to the implementation does not require a client to be re-compiled.

On the other hand, Cyclone generics do not provide the full expressive power of C++ templates. Rather, the support for generics in Cyclone is closer to what is found in languages such as ML, Haskell and does not require the implementation to duplicate generated code for different instantiations. Cyclone does not provide classes or objects. Instead, the other features in the language can be used to encode or simulate these features, just as in C. Unlike C or C++, these orthogonal features combine in a way that continues to enjoy type safety.

Interoperability and Portability

One other important feature in Cyclone is that it provides an escape hatch. It is possible to mix C code into a Cyclone program, just as it is possible to add native code to the Java VM. And in both settings, the safety of the system can be compromised by bad "native" code. However, real applications need to link against legacy libraries and talk with other languages. We have taken great pains to ensure that Cyclone code can call into C code and vice versa with a minimum of hassle. Since every C type is also a Cyclone type, there is no need for a separate foreign function interface. Futhermore, Cyclone retains the same calling conventions and data representations for those types that it has in common with C. This helps to minimize coercions as we cross the C/Cyclone boundary. Finally, by using a conservative garbage collector that is compatabile with C, we avoid the need for registering pointers or many of the other headaches needed to interface C code with other safe languages.

The Cyclone compiler generates C code that is then fed into GCC which makes the language relatively portable. We have developed a separate tool, called `buildlib` to make it easier to port system-specific header files and libraries to the language. At this point, we have ports for a number of platforms including Linux, Mac OS X, Win32/Cygwin, and even a few embedded systems such as the Lego Mindstorm platform.

The Cyclone distribution includes the full sources of the compiler (which is itself written in Cyclone) as well as a number of libraries and tools. For instance, we ported Bison and Flex to Cyclone so that we could use them to construct the front-end of the compiler.

We have also ported a number of other systems applications, such as web and ftp servers or Linux kernel modules to Cyclone. A small number of third parties are using the language in their own research projects.

Summary and Conclusions

At this point, you might be wondering whether Cyclone is suitable for your project. If you're committed to safety, or simply curious, then the answer is definitely! On the other hand, the language is a research project and as a result is still evolving. So if you're trying to get a product out the door, you might be better served going a more traditional route.

Our current research goals are to make it easier to port existing C code to Cyclone, to provide even stronger safety and security guarantees, and to eliminate as much performance overhead as possible. To that end, we would appreciate your feedback and criticisms.

Learning More

We have mailing lists, documentation, license information, benchmarks, technical papers, etc. at `www.research.att.com/projects/cyclone/`.