# Cyclone: A safe dialect of C

Trevor Jim* Greg Morrisett† Dan Grossman† Michael Hicks†
James Cheney† Yanling Wang†

November 20, 2001

**Abstract**

Cyclone is a safe dialect of C. It has been designed from the ground up to prevent the buffer overflows, format string attacks, and memory management errors that are common in C programs, while retaining C's syntax and semantics. This paper examines safety violations enabled by C's design, and shows how Cyclone avoids them, without giving up C's hallmark control over low-level details such as data representation and memory management.

## 1 Introduction

It is a commonly held belief in the security community that safety violations such as buffer overflows are unprofessional and even downright sloppy. This recent quote [27] is typical:

> Common errors that cause vulnerabilities — buffer overflows, poor handling of unexpected types and amounts of data — are well understood. Unfortunately, features still seem to be valued more highly among manufacturers than reliability.

The implication is that safety violations can be prevented just by changing priorities.

It's true that highly trained and motivated programmers can produce extremely robust systems when security is a top priority (witness OpenBSD). It's also true that most programmers can and should do more to ensure the safety and security of the programs that they write. However, we believe that the reasons that safety violations show up so often in C programs reach deeper than just poor training and effort: they have their roots in the design of C itself.

Take buffer overflows, for example. Every introductory C programming course warns against them and teaches techniques to avoid them, yet they continue to be announced in security bulletins every week. There are reasons for this that are more fundamental than poor training:

- One cause of buffer overflows in C is bad pointer arithmetic, and arithmetic is tricky. To put it plainly, an off-by-one error can cause a buffer overflow, and we will never be able to train programmers to the point where off-by-one errors are completely eliminated.

- C uses NUL-terminated strings. This is crucial for efficiency (a buffer can be allocated once and used to hold many different strings of different length before deallocation), but there is always a danger of overwriting the NUL terminator, usually leading to a buffer overflow in a library function. Some library functions (`strcat`) have an alternate version (`strncat`) that helps, by letting the programmer give a bound on the length of a string argument, but there are many dozens of functions in POSIX that do not have such an option.

*AT&T Labs Research, trevor@research.att.com
†Cornell University, http://www.cs.cornell.edu/projects/cyclone

- Out-of-bounds pointers are commonplace in C. The standard way to iterate over the elements of an array is to start with a pointer to the first element and increment it until it is just past the end of the array. This is blessed by the C standard, which states that the address just past the end of any array must be a valid address. When out-of-bounds pointers are common, you have to expect that occasionally one will be dereferenced or assigned, causing a buffer overflow.

In short, the design of the C programming language encourages programming at the edge of safety. This makes programs efficient but also vulnerable, and leads us to conclude that safety violations are likely to remain common in C programs. A number of studies bear this out [19, 9, 22, 16].

If C programs are unsafe, it's tempting to suggest that all programs be written in a safe language like Java (or ML, or Modula-3, or even 40-year-old Lisp). However, this isn't a realistic solution for everyone. For one thing, it abandons legacy code. For another, all of the safe languages look very different from C: they are high-level and abstract, they do not have explicit memory management, and they do not give programmers control over low-level data representations. These features make C unique, efficient, and indispensable to systems programmers.

We are developing an alternative for those who want safety but do not want to switch to a high-level language: Cyclone, a dialect of C that has been tweaked to prevent safety violations. Our goal is to design Cyclone so that it has the safety guarantee of Java (that no valid program can commit a safety violation) while keeping C's syntax, types, semantics, and programming idioms intact. In Cyclone, as in C, programmers can "feel the bits." We think that C programmers will have little trouble adapting to our dialect and will find Cyclone to be an appropriate language for many of the problems that ask for a C solution.

Cyclone has been in development for two years. We have written the Cyclone compiler in Cyclone (40,000 lines of code), and we've ported an additional 30,000 lines of C code to Cyclone, including tools like the Bison parser generator, a small web server, and many benchmarks. Cyclone is freely available from our web site, and comes with extensive documentation. The compiler itself and most of the accompanying tools are licensed under the GNU General Public License, and most of the libraries are licensed under the GNU LGPL.

This paper presents the design philosophy behind Cyclone, gives an overview of the techniques we've used to make a safe version of C, and reviews the history of the project, the mistakes we've made, and the course corrections that they inspired. We have written a separate paper [18], available from our web site, that discusses some of our techniques in more detail, giving some formalism, proofs of correctness, and detailed benchmarks.

The remainder of the paper is organized as follows. Section 2 points out some of the features of C that can lead to safety violations, and describes the changes we've made to prevent this in Cyclone. Section 3 gives some details about our implementation and its performance. Section 4 discusses the evolution of Cyclone's design, pointing out key decisions that we made and mistakes that we later reversed. We discuss future work in Section 5. In section 6, we discuss existing approaches to making C safer, and explain how Cyclone's approach is different. We conclude in Section 7.

# 2 From C to Cyclone

The Cyclone compiler performs a static analysis on source code, and inserts runtime checks into the compiled output at places where the analysis cannot determine that an operation is safe. The compiler may also refuse to compile a program. This may be because the program is truly unsafe, or may be because the static analysis is not able to guarantee that the program is safe, even by inserting runtime checks. If the compiler rejects a safe program, the programmer must modify the program, essentially providing extra information to the analysis so that it can verify safety.

Cyclone can thus be described as a superset of a subset of C. We throw out some programs that a C compiler would happily compile: this includes all of the unsafe C programs as well as some perfectly safe programs. We must reject some safe programs, because it is impossible to implement an analysis that perfectly separates the safe programs from the unsafe programs. We have developed a set of language extensions that programmers can use in their programs to help our analysis verify safety, and we have tried to design the extensions so that programmers need to make few modifications to get a safe C program to pass our analysis.

Some of the methods we use have been applied in other projects. The difference is that Cyclone provides far greater coverage of safety violations than previous work; our goal is to reach the level of safety provided by safe high-level languages like ML and Java. Previous approaches have only applied these techniques to C in a haphazard manner. We return to this point in Section 6.

Exactly how Cyclone works is best explained by example. In the rest of this section, we give examples of safety violations in C code, show how Cyclone detects them, and describe our language extensions. Some of the safety violations, like buffer overflows, can lead to root exploits. All of them can lead to crashes, which can be exploited to mount denial of service attacks [1, 10, 13, 21, 14].

**NULL** Consider the getc function:

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

If you call `getc(NULL)`, what happens? The C standard gives no definitive answer. If `getc` is written with safety in mind, it will perform a NULL check on its argument. That would be inefficient in the common case, though, so the check is probably omitted, leading to a segmentation fault.

Cyclone provides two solutions. The first is to automatically insert runtime NULL checks when pointers are used. For example, Cyclone will insert a NULL check in the body of `getc` when its argument is dereferenced.

This requires little effort from the programmer, but the NULL checks slow down `getc`. To repair this, we have extended Cyclone with a new kind of pointer, called a "never-NULL" pointer, and indicated with '`@`' instead of '`*`'. For example, in Cyclone you can declare

```
int getc(FILE @);
```

indicating that `getc` expects a non-NULL FILE pointer as argument. This one-character 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);         // Cyclone inserts a NULL check here
```

Cyclone prints a warning when it inserts the NULL check. This can be suppressed with an explicit cast:

```
getc((FILE @)f); // No warning, the cast is an explicit NULL check
```

A programmer can force the NULL check to occur only once by declaring a new `@`-pointer variable, and using the new variable at each call:

```
FILE @g = (FILE @)f;  // NULL check here
getc(g);              // No NULL check
```

Finally, constants like `stdin` are declared as `@`-pointers in the first place, and functions can be declared to return `@`-pointers. The effect is that NULL checks can be pushed back from their uses all the way to their sources. This is just as in C, except that in Cyclone, the compiler can ensure that NULL dereferences do not occur.

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.

**Buffer overflows** To prevent buffer overflows, we restrict pointer arithmetic: Cyclone does not permit pointer arithmetic on `*`-pointers or `@`-pointers. Instead, we provide another kind of pointer, indicated by '`?`', which permits pointer arithmetic. A `?`-pointer is represented by an address plus bounds information; since the representation of a `?`-pointer takes up more space than a `*`-pointer or `@`-pointer, we call it a "fat" pointer. The extra information in a fat pointer allows Cyclone to determine the size of the array pointed to, and to insert bounds checks at pointer accesses to ensure safety.

Here's an example of fat pointers in use — the string length function written in Cyclone:

```
int strlen(const char ?s) {
  int i, n;
  if (!s) return 0;
  n = s.size;
  for (i = 0; i < n; i++,s++)
    if (!*s) return i;
  return n;
}
```

There are only two differences between the Cyclone `strlen` and the C version. First, we declare the argument `s` to be a fat pointer to `char`, rather than a `*`-pointer. Second, in the body of the function we are able to get the size of the array pointed to by `s`, using the notation `s.size`. This lets us check that `s` is in-bounds in the for loop. That means we are guaranteed that we will never dereference `s` outside the bounds of the string, even if the NUL terminator is missing. In contrast, the C `strlen` will scan past the end of a string that lacks a NUL terminator.

Fat pointers add overhead to programs, because they take up more space than other pointers, and because of inserted bounds checks. However, they ensure safety, they give the programmer new capabilities (finding the size of the base array), and the programmer has explicit control over where they are used. It's easy to use `?`-pointers in Cyclone. A programmer who wants to use a `?`-pointer only needs to change a single character ('`*`' to '`?`') in a declaration. Arrays and strings are converted to `?`-pointers as necessary (automatically by the compiler). A programmer can explicitly cast a `?`-pointer to a `*`-pointer (this inserts a bounds check) or to a `@`-pointer (this inserts a NULL check and a bounds check). A `*`-pointer or `@`-pointer can be cast to a `?`-pointer, without any checks.

**Uninitialized pointers** The following snippet of C crashed one author's Palm Pilot:

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

This is part of a function that processes events. The problem is that while the pointer `f` is properly initialized in the first case of the switch, it is (by oversight) not initialized in the second case. So when the function `FrmGetObjectIndex` dereferences `f`, it isn't accessing a valid pointer, but rather some random address — whatever was on the stack when the space for `f` was allocated.

To prevent this in Cyclone, we perform a control-flow analysis on the source code. The analysis detects that f might be uninitialized in the second case, and the compiler signals an error. Usually, this catches a real bug, but there are times when our analysis isn't smart enough to figure out that something is properly initialized. This may force the programmer to initialize variables earlier than in C.

We don't consider it an error if non-pointers are uninitialized. For example, if you declare a local array of non-pointers, you can use it without initializing the elements:

```
char buf[64];       // buf contains garbage ...
sprintf(buf,"a");  // ... but there's no error here ...
char c = buf[20];  // ... or even here
```

This is common in C code, and does not compromise safety.

**Dangling pointers** Here is a naive (unsafe!) version of a C function that takes an int and returns its string representation:

```
char *itoa(int i) {
  char buf[20];
  sprintf(buf,"%d",i);
  return buf;
}
```

The function allocates a character buffer on the stack, prints the int into the buffer, and returns a pointer to the buffer. The problem is that the caller now has a pointer into deallocated stack space; this can easily lead to safety violations.

Cyclone prevents dangling pointers by performing a *region analysis* on the code. A region is a segment of memory that is deallocated all at once. For example, Cyclone considers all of the local variables of a block to be in the same region, which is deallocated on exit from the block. Cyclone's region analysis keeps track of what regions are live at any point in the program, and does not allow the use of any pointer into a non-live region. In the example above, Cyclone does not allow a pointer to `buf` to be returned from the function, because the region holding `buf` is going to be deallocated on the return.

**Free** C's `free` function can create dangling pointers, and it can segfault if it's called on a pointer that wasn't returned from `malloc` [14]. It is difficult to design an analysis that can guarantee correct use of pointers and `free`, so our current solution is drastic: we make `free` a no-op. This ensures that `free` cannot segfault or create dangling pointers.

Obviously, programmers still need a way to reclaim heap-allocated data. We provide two ways. First, the programmer can use an optional garbage collector. This is very helpful in getting existing C programs to port to Cyclone without many changes. However, in many cases it constitutes an unacceptable loss of control.

We recognize that C programmers need explicit control over allocation and deallocation. Therefore, Cyclone provides a feature called *growable regions*. The following code declares a growable region, does some allocation into the region, and deallocates the region:

```
region h {
  int *x = rmalloc(h,sizeof(int));
  int ?y = rnew(h) { 1, 2, 3 };
  char ?z = rprintf(h,"hello");
}
```

The code uses a `region` block to start a new, growable region that lives on the heap. The region is deallocated on exit from the block. The variable `h` is a *handle* for the region and it is used to allocate into the region, in one of several ways.

First, there is an `rmalloc` function that behaves like `malloc` except that it requires a region handle as an argument; it allocates into the region of the handle. In the example above, `x` is initialized with a pointer to an int-sized chunk of memory allocated in `h`'s region.

Second, the `rnew` construct is used when the programmer wants to allocate and initialize in a single step. For example, `y` is initialized above as a fat pointer to an array with elements 1, 2, and 3, allocated in `h`'s region.

Finally, region handles may be passed to functions, like the library function `rprintf`. `rprintf` is like `printf`, except that it does not print to a file; instead it allocates a buffer in a region, places the formatted output in the buffer, and returns a pointer to the buffer. In the example above, `z` is initialized with a pointer to the string "hello" that is allocated in `h`'s region.

Our region analysis knows that `x`, `y`, and `z` all point into `h`'s region, and that the region is deallocated on exit from the block. It uses this knowledge to prevent dangling pointers into the region — for example, it prohibits storing `x` into a global variable, which could be used to (wrongly) access the region after it is deallocated.

Growable regions are a safe version of arena-style memory management, which is widely used (e.g., in Apache). C programmers use many other styles of memory management, and we plan in the future to extend Cyclone to accommodate more of them safely. In the meantime, Cyclone is one of the very few safe languages that supports safe, explicit memory management, without relying on a garbage collector.

**Type-varying arguments** In C it is possible to write a function that takes an argument whose type varies from call to call. The most obvious example is `printf`:

```
printf("%d", 3); printf("%s", "hello");
```

In the first call to `printf`, the second argument is an `int`, and in the next call, the second argument is a `char *`. This is perfectly safe in this case, and the compiler can even catch errors by examining the format string to see what types the remaining arguments should have. Unfortunately, the compiler can't catch all errors. Consider:

```
extern char *y; printf(y);
```

This is a lazy way to print the string `y`. The problem is that, in general, `y` can contain `%` format directives, causing `printf` to look for non-existent arguments on the stack. The compiler can't check this because `y` is not a string literal. A core dump is not unlikely.

The danger is greater if the user of the program gets to choose the string `y`. There is a format directive, `%n`, that causes `printf` to write the number of characters printed so far into a location specified by a pointer argument; this can be used to write an arbitrary value to a location chosen by the attacker, leading to a complete compromise. This is known as a format string attack, and it's an increasingly common exploit [28].

We solve this in Cyclone in two steps. First, we add *tagged unions* to the language:

```
tunion t {
  Int(int);
  Str(char ?);
};
```

This declares a new tagged union type, `tunion t`. A tagged union has several cases, like an ordinary union, but adds tags that distinguish the cases. Here, `tunion t` has an int case with tag

Int, and a char ? case with tag Str. A function that takes a tagged union as argument can look at the tags to find out what case the argument is in, using an extension of the switch statement:

```
void print(tunion t x) {
  switch (x) {
  case &Int(i): printf("%d",i); break;
  case &Str(s): printf("%s",s); break;
  }
}
```

The first case of the switch will be executed if x has tag Int; the variable i gets bound to the underlying int, so it can be used in the body of the case. Similarly, the second case is taken if x has tag Str with underlying string s.

Tags enable the print function to correctly detect the type of its argument. However, callers have to explicitly add tags to the arguments. For example, print can be called as follows:

```
print(new Int(4));
print(new Str("hello"));
```

The first line calls print with the int 4, adding the tag Int with the notation new Int(4). The second call does the same with string "hello" and tag Str.

Inserting the tags by hand is inconvenient, so we also provide a second feature, *automatic tag injection*. For example, in Cyclone printf is declared

```
printf(char ?fmt, ... inject tunion printfargs);
```

where printfargs is a tagged union containing all of the possible types of arguments for printf. Cyclone's printf is called just as in C, without explicit tags:

```
printf("%s %i", "hello", 4);
```

The compiler inserts the correct tags automatically (they are placed on the stack). The printf function itself accesses the tagged arguments through a fat pointer (Cyclone's varargs are bounds checked) and uses switch to make sure the arguments have the right type; this makes printf safe even if the format string argument comes from user input.

Type-varying arguments are used in many other POSIX functions, including the scanf functions, fcntl, ioctl, signal, and socket functions such as bind and connect. Tagged unions and injection allow us to make sure these functions are called safely, while presenting the same interface to the programmer.

**Other vulnerabilities** These are only a few of the features of C that can be misused to cause safety violations. Other examples are: bad casts; gotos between scopes; varargs (as implemented in C); missing return statements; violations of const qualifiers; and improper use of unions. Cyclone's analysis restricts these features to prevent safety violations.

# 3 Implementation

The Cyclone compiler is implemented in approximately 40,000 lines of Cyclone. It consists of a parser, a static analysis phase, and a simple translator to C. We use gcc as a back end and have also experimented with using VC++. We are able to use some existing tools (gdb, flex) and we ported others completely to Cyclone (bison). When a user compiles with garbage collection enabled, we use the Boehm-Demers-Weiser conservative garbage collector as an off-the-shelf component. We have also built some useful utilities, including a documentation generation tool and a memory profiler.

| Program | LOC | | diffs | | performance | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | C | Cyc | + | − | C time (s) | checked(s) | % | unchecked(s) | % |
| cacm | 340 | 359 | 42 | 23 | 1.77 | 3.49 | 97% | 3.03 | 71% |
| cfrac | 4218 | 4214 | 132 | 136 | 2.61 | 17.07 | 554% | 17.07 | 554% |
| finger | 158 | 161 | 18 | 15 | 0.58 | 0.55 | -5% | 0.48 | -17% |
| grobner | 3244 | 3377 | 438 | 305 | 0.07 | 0.20 | 186% | 0.20 | 186% |
| http_get | 529 | 529 | 36 | 36 | 0.28 | 0.28 | 0% | 0.28 | 0% |
| http_load | 2072 | 2057 | 115 | 130 | 89.37 | 90.22 | 1% | 90.19 | 1% |
| http_ping | 1072 | 1081 | 30 | 21 | 0.28 | 0.28 | 0% | 0.28 | 0% |
| http_post | 607 | 608 | 42 | 41 | 0.16 | 0.16 | 0% | 0.16 | 0% |
| matxmult | 57 | 48 | 3 | 12 | 1.38 | 1.83 | 32% | 1.38 | 0% |
| mini_httpd | 3005 | 3022 | 233 | 216 | 3.71 | 3.85 | 4% | 3.86 | 4% |
| ncompress | 1964 | 1982 | 120 | 102 | 0.20 | 0.39 | 95% | 0.38 | 90% |
| tile | 1345 | 1366 | 145 | 124 | 0.48 | 1.05 | 116% | 0.99 | 104% |
| total | 18611 | 18804 | 1354 | 1161 | - | - | - | - | - |
| *"regionized" versions of benchmarks* | | | | | | | | | |
| cfrac | 4218 | 4110 | 501 | 528 | 2.61 | 10.07 | 286% | 8.80 | 237% |
| mini_httpd | 3005 | 2967 | 500 | 522 | 3.71 | 3.83 | 3% | 3.82 | 3% |
| total | 7223 | 7174 | 1001 | 1050 | - | - | - | - | - |

Table 1: Benchmarks

In order to get a rough idea of the current and potential performance of the language, we ported a selection of benchmarks from C to Cyclone. Surprisingly, the benchmarks were useful in evaluating Cyclone's safety as well as its performance: several of the benchmarks had safety violations that were revealed when we ported them to Cyclone.

**The benchmarks** We tried to pick benchmarks from a range of problem domains. For networking, we used the `mini_httpd` web server; the web utilities `http_get`, `http_post`, `http_ping`, and `http_load`; and `finger`. `cfrac`, `grobner`, `tile`, and `matxmult` are computationally intensive C applications that make heavy use of arrays and pointers. Finally, `cacm` and `ncompress` are compression utilities.

**Ease of porting** We've tried to design Cyclone so that existing C code can be ported with few modifications. Table 1 quantifies the number of modifications we needed to port the benchmarks. For each benchmark, the table shows the number of lines of code in both the C and Cyclone versions, and the number of lines changed according to diff. In porting the first grouping of benchmarks, we tried to minimize changes. The second grouping gives results for benchmarks that we modified more heavily, for performance sake, primarily by using Cyclone's growable regions.

In these benchmarks, the changes required to get them to compile in Cyclone were small: less than 10% of the lines needed to be changed. Most of the changes were one-character changes dealing with pointers, for example, changing `char *` to `char ?`. Most of the other changes had to do with allocation.

**Performance** Table 1 also gives performance numbers. We have not yet implemented standard bounds-check elimination techniques in the compiler, because our effort to date has focused on safety, rather than performance. For this reason, we show results for Cyclone with runtime checks enabled, and with no runtime checks inserted.

The results are median running times (n=21) on a 750 MHz Pentium III with 256MB of RAM, running Linux kernel 2.2.16-12. The percentages give the overhead of Cyclone over C. We achieve near-zero overhead for I/O bound applications such as the web server, but there is a considerable overhead for computationally-intensive benchmarks, up to 3x for the benchmarks here. In other micro benchmarks not listed here, we have seen up to 10x overhead. Bounds checks are a major component of the overhead, as can be seen by comparing the checked and unchecked times for `matxmult`.

**Safety** Perhaps the most interesting thing we learned by porting these benchmarks from C is that several of them had safety violations, despite being used as benchmarks in other papers. `grobner`, whose code dates as far back as 1984, was particularly egregious. Nearly all of the bugs had to do with array bounds violations.

The `mini_httpd` web server consults a file, `.htpasswd`, to decide whether to grant client access to protected web pages. It tries to be careful not to reveal the password file to clients. Ironically, the code to protect the password file contains a safety violation:

```
#define AUTH_FILE ".htpasswd"
... strcmp( &(file[strlen(file) - sizeof(AUTH_FILE) + 1]),
            AUTH_FILE ) == 0 ...
```

The code is trying to see if the file requested by the client is `.htpasswd`. Unfortunately, if `file` is a string shorter than `.htpasswd`, then `strcmp` will be passed an out-of-bounds pointer. This can prevent legitimate files from being accessed. Cyclone caught the error with a runtime bounds check.

We found a less innocent bounds violation in `grobner`. It represents polynomials as arrays of coefficients, and has a multiply routine that handles polynomials with a single coefficient as a special case. Unfortunately, the code for the general case turns out to be completely wrong: a loop is unrolled incorrectly, and the multiplication ends up being applied to out-of-bounds pointers. As a result, the answers returned are essentially non-deterministic. Given `grobner`'s age, it is surprising that this has not been caught before; four of the ten test cases provided in the distribution follow this code path. In Cyclone, our bounds checks quickly illuminated the source of the problem.

`tile` contained a bounds error due to an order-of-evaluation bug in this code:

```
mksentarrays(cur_sentsize, cur_sentsize += GROWSENT);
```

The function `mksentarrays` reallocates a global array. The old size of the array should be the first argument, and the new size should be the second argument. Order of evaluation happened to be right-to-left on our platform, so this code passes the new size of the array to `mksentarrays` in the first argument. This caused a bounds exception in Cyclone. The C version used the same evaluation order, but the out-of-bounds access was not caught (this caused an incorrect initialization of the new array).

# 4 Design history

Cyclone began as an offshoot of the Typed Assembly Language (TAL) project [24]. The TAL project's goal was to ensure program safety at the machine code level, by adding machine-checkable safety annotations to machine code. The machine code annotations are not easy to produce by hand, so we designed a simple, C-like language called Popcorn as a front end, and built a compiler that automatically translates Popcorn to machine code plus the necessary annotations.

Popcorn worked out well as a proof-of-concept for TAL, but it had some disadvantages. It was C-like, but different enough to make porting C code and interfacing to C code difficult. It was also a language that was only used by our own research group, and was unlikely to be adopted by anyone else. Cyclone is a reworking of Popcorn with two agendas: to further our understanding of low-level safety, and to gain outside adopters.

It turns out that taking C compatibility as a serious requirement was critical to advancing both of these agendas. It was obvious from the start that C compatibility would make Cyclone more appealing to others, but the idea that it would help us to understand how to better design a *safe* low-level language was a surprise.

C programmers don't write the same kinds of programs as Java programmers or ML programmers. They use many tricks that aren't available in high-level languages. While many C programs are not 100% safe, most are intended to be safe, and we learned a great deal from porting systems code from C to Cyclone. Often, we found that we had made choices in the design of Cyclone that were holdovers from ML [23], another language that we had worked on. Some (most!) of these choices were right for ML, but not for C, or for Cyclone, and we ended up following C more closely than we had expected at the start.

All of this has played out gradually over the years of Cyclone's development. Here are some of the more notable mistakes and course changes we've made:

- Originally, we supported arrays not with fat pointers, but with a type `array<t>`, where `t` is the element type of the array. An `array<t>` could be passed to functions, and a value of type `array<t>` supported subscripting, but not pointer arithmetic. This matches up closely with ML's array types, and was a carryover from when Popcorn was implemented in ML. However, converting C code to use `array<t>` was painful, requiring nontrivial editing of type declarations, and converting pointer arithmetic to array subscripting. We abandoned it for fat pointers, which make it easy to port C code, requiring only a few changes from '`*`' to '`?`', and no changes to pointer arithmetic.

- We didn't understand the importance of NUL-terminated strings. NUL termination isn't guaranteed in C, so, for safety, we were committed to using explicit array bounds from the beginning. The NUL seemed pointless, and our first string library ignored it. As we programmed more in the language and ported C code, we came to understand how important NUL is to efficiency (memory reuse), and we changed our string library to match up with C's.

- In C, a switch case by default falls through to the next case, unless there is an explicit break. This is exactly the opposite of what it should be: most cases do not fall through, and, moreover, when a case does fall through, it is probably a bug. Therefore, we added an explicit fallthru statement, and used the rule that a case would NOT fall through unless the fallthru statement was used.

  Our decision to "correct" C's mistake was wrong. It made porting error-prone because we had to examine every switch statement to look for intentional fall throughs, and add a fallthru statement. We had also gotten rid of any special meaning of break within switch, since it was no longer needed — consequently, a break in a switch within a loop would break to the head of the loop (in early versions of Cyclone). Eventually, we realized that we were going against a basic instinct of every C programmer, without gaining much of anything, so we restored C's semantics of switch and break.

- We originally implemented tagged unions as an extension of enums, since an enum constant is like a case of a tagged union with no associated value. Since a tagged union is more general, we decided to just have one of the two.

  This was a mistake because in C, an enum type is really treated as int, and C programmers rely on this. It's not uncommon to see things like

  ```
  x = (x+1)%3;
  ```

where x is a variable with enum type. We aren't able to do this with tagged unions, so we eventually separated them from enum.

# 5 Future work

C programmers use a wide variety of memory management strategies, but at the moment, Cyclone only supports garbage collection and arena memory management. A major goal of the project going forward will be to research ways to accommodate other memory management strategies, while retaining safety.

Another limitation of our current release is that its implementation of run-time checks is not thread-safe: for example, we must prevent the possibility that a fat pointer can be mutated in between the bounds check and the ensuing dereference. Copying a fat pointer before doing the check and dereference suffices, but we would prefer to incorporate a static analysis that avoids the copy for values that are thread-local. Better yet would be exposing whether values are thread local so that programmers could control any performance overhead associated with shared values.

# 6 Related work

There is an enormous body of research on making C safer. Most techniques can be grouped into one of the following strategies:

1. Static analysis. Programs like LINT crawl over C source code and flag possible safety violations, which the programmer can then review. Just a few other examples are LCLint [15, 20], Metal [11, 12]. SLAM [4, 3], PREfix [6], and cqual [26].

2. Inserting runtime checks. C's assert statements, the Safe-C system [2], and "debugging" versions of libraries, like Electric Fence, cause programs to perform sanity checks as they run. This technique has been used to combat buffer overflows [8, 5, 17] and printf format string attacks [7].

3. Combining static analysis and runtime checks. Systems like CCured [25] perform static analyses to check source code for safety, and automatically insert runtime checks where safety cannot be guaranteed statically.

These are good techniques — Cyclone itself uses the third strategy. However, except for CCured, none of the above projects applies them in a way that comes close to ruling out all of the safety violations found in C. It is not hard for a program to pass LINT and still crash, and even the more advanced checking systems, like LCLint, SLAM, and Metal, do not find all safety violations. We can say something similar about all of the other systems mentioned above. Furthermore, most of these systems are simply not used — assert is probably the most popular, but it is usually turned off when code is shipped to avoid performance degradation.

CCured and Cyclone both seek to rule out all safety violations. The main disadvantage of CCured is that it takes control away from programmers. CCured needs to maintain some extra bookkeeping information in order to perform necessary runtime checks, and it does this by modifying data representations. For example, an `int *` might be represented by just an address, but it might also be represented by an address plus extra data that allows bounds checking. This means that CCured has control over data representations, not the programmer; and, moreover, basic operations (dereferencing, pointer arithmetic) will have different costs, depending on the decisions made by CCured. Furthermore, CCured relies on a garbage collector, so programmers have less control over memory management.

## 7 Conclusion

Cyclone is a C dialect that prevents safety violations in programs using a combination of static analyses and inserted runtime checks. Cyclone's goal is to accommodate C's style of low-level programming, while providing the same level of safety guaranteed by high-level safe languages like Java — a level of safety that has not been achieved by previous approaches.

## References


[1] Denial-of-service attack via ping. CERT Advisory CA–1996–26, December 18, 1996. http://www.cert.org/advisories/CA-1996-26.html.

[2] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *ACM Conference on Programming Language Design and Implementation*, pages 290–301, June 1994.

[3] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 203–213, June 2001.

[4] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, May 2001.

[5] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack-smashing attacks. In *USENIX Annual 2000 Technical Conference*, San Diego, California, June 2000.

[6] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software, Practice, and Experience*, 30(7):775–802, 2000.

[7] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *10th USENIX Security Symposium*, Washington, D.C., August 2001.

[8] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, San Antonio, Texas, January 1998.

[9] John DeVale and Philip Koopman. Performance evaluation of exception handling in I/O libraries. In *The International Conference on Dependable Systems and Networks*, June 2001.

[10] Roman Drahtmueller. Re: SuSE Linux 6.x 7.0 Ident buffer overflow. `Bugtraq` mailing list, November 29, 2000. http://www.securityfocus.com/archive/1/147592.

[11] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation*, October 2000.

[12] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of Eighteenth ACM Symposium on Operating Systems Principles*, October 2001.

[13] Chris Evans. "gdm" remote hole. `Bugtraq` mailing list, May 22, 2000. http://www.securityfocus.com/archive/1/61099.

[14] Chris Evans. Very interesting traceroute flaw. `Bugtraq` mailing list, September 28, 2000. http://www.securityfocus.com/archive/1/136215.

[15] David Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, May 1996.

[16] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows Systems Symposium*, August 2000.

[17] Mike Frantzen and Mike Shuey. Stackghost: Hardware facilitated stack protection. In *10th USENIX Security Symposium*, Washington, D.C., August 2001.

[18] Dan Grossman, Greg Morrisett, Trevor Jim, Mike Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. Draft manuscript.

[19] Philip Koopman and John DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering*, 26(9), September 2000.

[20] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, Washington, D.C., August 2001.

[21] Elias Levy. Re: rpc.ttdbserverd on solaris 7. `Bugtraq` mailing list, November 19, 1999. http://www.securityfocus.com/archive/1/35480.

[22] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of Unix utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[23] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[24] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52, Kyoto, Japan, March 1998. Springer-Verlag.

[25] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Twenty-Ninth ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002. To appear.

[26] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, Washington, D.C., August 2001.

[27] Stephan Somogyi and Bruce Schneier. Inside risks: The perils of port 80. *Communications of the ACM*, 44(10), October 2001.

[28] "tf8". Wu-Ftpd remote format string stack overwrite vulnerability. Bugtraq vulnerability 1387, June 22, 2000. http://www.securityfocus.com/bid/1387.