USENIX Association

Proceedings of the 2002 USENIX Annual Technical Conference

Monterey, California, USA
June 10-15, 2002

Cyclone: A safe dialect of C

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

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 [33] 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 lengths 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 alternate versions (`strncat`) that help, by letting the programmer give a bound on the length of a string argument, but there are many dozens of functions in POSIX with no such alternative.
- 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 valid. 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

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

to remain common in C programs. A number of studies bear this out [23, 11, 28, 18].

If C programs are unsafe, it is 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 is not 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 designed to prevent safety violations. Our goal is to design Cyclone so that it has the safety guarantee of Java (no valid program can commit a safety violation) while keeping C's syntax, types, semantics, and 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. In total, we have written about 110,000 lines of Cyclone code, with about 35,000 lines for the compiler itself, and 15,000 lines for supporting libraries and tools, like a port of the Bison parser generator. We have also ported about 50,000 lines of benchmark applications, and are developing a streaming media overlay network in Cyclone [27]. Cyclone is freely available and comes with extensive documentation. The compiler 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 is a high-level overview of Cyclone. It 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.

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 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

Most of Cyclone's language design comes directly from C. Cyclone uses the C preprocessor, and, with few exceptions, follows C's lexical conventions and grammar. Cyclone has pointers, arrays, structures, unions, enumerations, and all of the usual floating point and integer types; and they have the same data representation in Cyclone as in C. Cyclone's standard library supports a large (and growing) subset of POSIX. The intention is to make it easy for C programmers to learn Cyclone, to port C code to Cyclone, and to interface C code with Cyclone code.

The major differences between Cyclone and C are all related to safety. The Cyclone compiler performs a static analysis on source code, and inserts run-time 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 run-time checks. We reject 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.

When Cyclone rejects a safe C program, the programmer may choose to rewrite the program so that our analysis can verify its safety. To make this easier, we have identified common C idioms that our static analysis cannot handle, and have added features to the language so that these idioms can be programmed in Cyclone with only a few modifications. These modifications typically include adding annotations that supply hints to the static analysis, or that cause the program to maintain extra information needed for run-time checks (e.g., bounds checks).

Table 1: Restrictions imposed by Cyclone to preserve safety

- `NULL` checks are inserted to prevent segmentation faults
- Pointer arithmetic is restricted
- Pointers must be initialized before use
- Dangling pointers are prevented through region analysis and limitations on `free`
- Only "safe" casts and unions are allowed
- `goto` into scopes is disallowed
- `switch` labels in different scopes are disallowed
- Pointer-returning functions must execute `return`
- `setjmp` and `longjmp` are not supported

Table 2: Extensions provided by Cyclone to safely regain C programming idioms

- Never-`NULL` pointers do not require `NULL` checks
- "Fat" pointers support pointer arithmetic with run-time bounds checking
- Growable regions support a form of safe manual memory management
- Tagged unions support type-varying arguments
- Injections help automate the use of tagged unions for programmers
- Polymorphism replaces some uses of `void *`
- Varargs are implemented with fat pointers
- Exceptions replace some uses of `setjmp` and `longjmp`

Cyclone can thus be understood by starting from C, imposing some restrictions to preserve safety, and adding features to regain common programming idioms in a safe way. Cyclone's restrictions are summarized in Table 1, and its extensions are summarized in Table 2.

Some of the techniques we use to make Cyclone safe have been applied to C before, and there has been a great deal of research on additional techniques that we do not use in Cyclone. However, previous projects have typically used only one or two techniques, resulting in incomplete coverage. For example, McGary's bounded pointers protect against some, but not all, array access violations [26], and StackGuard protects against some, but not all, buffer overflows [9]. Our goal with Cyclone is to prevent all safety violations. Moreover, previous projects have been presented as optional add-ons to C, so in practice they are seldom used in production code; Cyclone makes safety the default.

In the rest of this section, we illustrate Cyclone's features by giving examples of safety violations in C code, explaining how Cyclone's restrictions detect and prevent them, and introducing the language extensions that can be used to safely program around the restrictions. Some of the safety violations we describe, 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 [6, 7, 12, 15, 25, 16].

`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 run-time `NULL` checks when pointers are used. For example, Cyclone will insert code into the body of `getc` to do a `NULL` check 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 its 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);            // NULL check here
```

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

```
getc((FILE @)f);  // Check w/o warning
```

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;
}
```

This looks like a C version of `strlen`, with two exceptions. 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; the resulting `?`-pointer has size 1.

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 ini-

tialized 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 an unpredictable address — whatever was on the stack when the space for `f` was allocated.

To prevent this in Cyclone, we perform a static 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];     // contains garbage ..
sprintf(buf,"a"); // .. but no err here
char c = buf[20]; // .. or even here
```

This is common in C code; since these array accesses are in-bounds, we allow them.

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.

It is easy for a C compiler to warn against returning the address of a local variable, and, indeed, `gcc` prints just such a warning for the example above. However, this technique will not catch even the following simple variation:

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

Here, the address of `buf` is stored in the variable `z`, and then `z` is returned. This passes `gcc -Wall` without complaint.

Cyclone prevents the dereference of 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 static region analysis keeps track of what region each pointer points into, and what regions are live at any point in the program. Any dereference of a pointer into a non-live region is reported as a compile-time error.

In this last example, Cyclone's region analysis knows that the address of `buf` is a pointer into the local stack of `itoa`. The assignment to `z` tells Cyclone that `z` is also a pointer into `itoa`'s stack area. Since the local stack area will be deallocated when `z` is returned from `itoa`, we report an error.

Cyclone's region analysis is intraprocedural — it is not a whole-program analysis. We rely on programmer annotations to track regions across function calls. For example, the `strcat` function is declared as follows in Cyclone:

```
char ?`r strcat(char ?`r dest,
                const char ? src);
```

Here `` `r `` is a *region variable*. The declaration says that for any region `` `r ``, `strcat` takes a pointer `dest` into region `` `r ``, and a pointer `src`, and returns a pointer into region `` `r ``. (In fact, the C standard specifies that `strcat` returns `dest`.) This information enables Cyclone to correctly reject the following program:

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

The region analysis deduces that the result of the call to `strcat` on `buf` points into the local stack region of `itoa`, so it cannot be returned from the function.

Cyclone's region analysis is described in greater detail in a separate paper [21].

Free C's `free` function can create dangling pointers, and, depending on how it is implemented, can cause segmentation faults or even root compromises if used incorrectly (e.g., if it is called with a pointer not returned by `malloc` [16], or if it is used to reclaim the same block of memory twice [7]). It is difficult to design an analysis that can guarantee the correct use of pointers and `free`, so our current solution is drastic: we make `free` a no-op.

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 (without an explicit `free`). 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` construct 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 `sprintf`, except that it does not print to a fixed-sized buffer; 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. Unlike `sprintf`, there is no risk of a buffer overflow, and unlike `snprintf`, there is no risk of passing a buffer that is too small. Moreover, the allocated buffer will be freed when the region goes out of scope, just as a stack-allocated buffer would be.

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 `printf` function is a familiar example:

```
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`. The `%n` format directive causes `printf` to write the number of characters printed so far into a location specified by a pointer argument; it 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 is an increasingly common exploit [34].

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 pr(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 `pr` function above to correctly detect the type of its argument. However, callers have to explicitly add tags to the arguments. For example, `pr` can be called as follows:

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

The first line calls `pr` 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 parg_t);
```

where `parg_t` 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 — Cyclone does not permit the `printf` programmer to use the arguments in a type-inconsistent way. Moreover, the tags let the programmer detect any inconsistency at run time and take appropriate action (e.g., return an error code or exit the program).

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`. Cyclone uses tagged unions and injection to make sure that these functions are called safely, while presenting the programmer with the same interface as in C.

Goto C's `goto` statements can lead to safety violations when they are used to jump into scopes. Here is a simple example:

```
int z;
{ int x = 0xBAD; goto L; }
{ int *y = &z;
  L: *y = 3;   // Possible segfault
}
```

The program declares a variable `z`, then enters two blocks in sequence. Many compilers stack allocate the local variables of a block when it is entered, and deallocate (pop) the storage when the block exits (though this is not mandated by the C standard). If the example is compiled in this way, then when the program enters the first block, space for `x` is allocated on the stack, and is initialized with the value 0xBAD. The `goto` jumps into the middle of the second block, directly to the assignment to the contents of the pointer `y`. Since `y` is the first (only) variable declared in the second block, the assignment expects `y` to be at the top of the stack. Unfortunately, that's exactly where `x` was allocated, so the program tries to write to location 0xBAD, probably triggering a segmentation fault.

Cyclone's static analysis detects this situation and signals an error. A `goto` that does not enter a scope is safe, and is allowed in Cyclone. We apply the same analysis to `switch` statements, which suffer from a similar vulnerability in C.

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; 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 35,000 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 Microsoft Visual C++. 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.

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. The benchmarks were useful in testing Cyclone's safety guarantees as well as its performance: several of the benchmarks had safety violations that were revealed (and we subsequently fixed) when we ported them to Cyclone. The process of porting also tested the limitations of Cyclone's interface to the C library and forced us to provide more complete library support. For example, even small benchmarks such as `finger` and `http_get` make use of parts of the C library that the Cyclone compiler and other tools do not, such as sockets and signals.

Program	LOC		diffs		
	C	Cyc	#	C %	? %
cacm	340	360	41	12%	0%
cfrac	4218	4215	134	3%	37%
finger	158	161	17	11%	12%
grobner	3260	3401	452	14%	24%
http_get	529	530	44	8%	45%
http_load	2072	2058	121	6%	24%
http_ping	1072	1082	33	3%	33%
http_post	607	609	51	8%	45%
matxmult	57	53	11	19%	9%
mini_httpd	3005	3027	266	9%	46%
ncompress	1964	1986	134	7%	25%
tile	1345	1365	148	11%	32%
total	18627	18847	1452	8%	31%
regionized benchmarks					
cfrac	4218	4192	503	12%	9%
mini_httpd	3005	2986	531	18%	24%
total	7223	7178	1034	14%	16%

Table 3: Benchmark diffs

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`. The `cfrac`, `grobner`, `tile`, and `matxmult` benchmarks are computationally intensive C applications that make heavy use of arrays and pointers. Finally, `cacm` and `ncompress` are compression utilities. All of the benchmark programs, in both C and Cyclone, can be found on the Cyclone homepage [10].

Ease of porting We have tried to design Cyclone so that existing C code can be ported with few modifications. Table 3 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. The diff # column shows the number of lines changed in each port, and the C % column shows the percentage of lines changed relative to the original program size. In porting the first grouping of benchmarks, we tried to minimize changes. In particular, the benchmarks involving non-trivial dynamic memory management (`cfrac`, `grobner`, `http_load`, and `tile`), were compiled with the garbage collector in Cyclone; all other benchmarks do not use the garbage collector. The second grouping gives results for versions of benchmarks that we modified to make use of Cyclone's growable regions wherever possible.

Usually fewer than 10% of the lines needed to be changed to port the benchmarks to Cyclone. One of the most common changes was changing C-style `*` pointers to Cyclone `?` pointers; for example, changing `char *` to `char ?`. The `?` % column of Table 3 shows the percentage of changes that were of this form: generally, this simple change accounted for 20–50% of changed lines. Most of the other changes had to do with adapting to Cyclone's stricter requirements for allocation, initialization, `const` enforcement, and function prototyping. Typical changes of these forms included changing `malloc` to `new`, adding explicit initializers, adding explicit `const` type qualifiers to casts, and ensuring that all functions have prototypes with explicit return values.

Performance Table 4 compares the performance of the benchmarks in C, in Cyclone with bounds checking enabled, and in Cyclone with bounds checking disabled. Presently we do only very simple bounds-check elimination, because our effort to date has focused on safety, rather than performance; the gap between the second and third measurements gives an upper bound for the improvement we can expect from this in the future.

We ran each benchmark twenty-one times on a 750 MHz Pentium III with 256MB of RAM, running Linux kernel 2.2.16-12, using `gcc` 2.96 as a back end. We used the `gcc` flags `-O3` and `-march=i686` for compiling all the benchmarks. Because we observed skewed distributions for the http benchmarks, we report medians and semi-interquartile ranges (SIQR).[1] For the non-web benchmarks (and some of the web benchmarks as well) the median and the mean were essentially identical, and the standard deviation was at most 2% of the mean.

The table also shows the slowdown factor of Cyclone relative to C. We achieve near-zero overhead for I/O bound applications such as the web server and the http programs, but there is a considerable overhead for computationally-intensive benchmarks; the worst is `grobner`, which is almost a factor of three slower than the C version. We have seen slowdowns of a factor of six in pathological scenarios involving pointer arithmetic in other microbenchmarks not listed here.

Two common sources of overhead in safe languages are garbage collection and bounds checking. The checked and unchecked columns of Table 4 show that bounds checks are an important component of our overhead, as expected. Garbage collection overhead is not as easy to measure. Profiling the garbage collected version of `cfrac` suggests that garbage collection accounts for approximately half of its overhead. Partially regionizing `cfrac` resulted in a 6% improvement with bounds checks on; but regionizing can require significant changes to the program, so the value of this comparison is not clear. We expect that the overhead will vary widely for different programs depending on their memory usage patterns; for example, `http_load` and `tile` make relatively little use of dynamic allocation, so they have almost no garbage collection overhead.

Cyclone's representation of fat pointers turned out to be another important overhead. We represent fat pointers with three words: the base address, the bounds address, and the current pointer location (essentially the same representation used by McGary's bounded pointers [26]). Compared to C's pointers, fat pointers have a larger space overhead, larger cache footprint, increased parameter passing overhead, and increased register pressure, especially on the register-impoverished x86. Good code generation can make a big difference: we found that using `gcc`'s `-march=i686` flag increased the speed of programs making heavy use of fat pointers (such as `cfrac` and `grobner`) by as much as a factor of two, because it causes `gcc` to use a more efficient implementation of block copy.

Safety We found array bounds violations in three benchmarks when we ported them from C to Cyclone: `mini_httpd`, `grobner`, and `tile`. This was a

[1] The semi-interquartile range is the difference between the high quartile and the low quartile divided by 2. This is a measure of variability, similar to standard deviation, recommended for skewed distributions [22].

Test	C time(s)	Cyclone time			
		checked(s)	factor	unchecked(s)	factor
cacm	0.12 ± 0.00	0.15 ± 0.00	1.25×	0.14 ± 0.00	1.17×
cfrac†	2.30 ± 0.00	5.57 ± 0.01	2.42×	4.77 ± 0.01	2.07×
finger	0.54 ± 0.42	0.48 ± 0.15	0.89×	0.53 ± 0.16	0.98×
grobner†	0.03 ± 0.00	0.07 ± 0.00	2.85×	0.07 ± 0.00	2.49×
http_get	0.32 ± 0.03	0.33 ± 0.02	1.03×	0.32 ± 0.06	1.00×
http_load†	0.16 ± 0.00	0.16 ± 0.00	1.00×	0.16 ± 0.00	1.00×
http_ping	0.06 ± 0.02	0.06 ± 0.02	1.00×	0.06 ± 0.01	1.00×
http_post	0.04 ± 0.01	0.04 ± 0.00	1.00×	0.04 ± 0.01	1.00×
matxmult	1.37 ± 0.00	1.50 ± 0.00	1.09×	1.37 ± 0.00	1.00×
mini_httpd-1.15c	2.05 ± 0.00	2.09 ± 0.00	1.02×	2.09 ± 0.00	1.02×
ncompress-4.2.4	0.14 ± 0.01	0.19 ± 0.00	1.36×	0.18 ± 0.00	1.29×
tile†	0.44 ± 0.00	0.74 ± 0.00	1.68×	0.67 ± 0.00	1.52×
†Compiled with the garbage collector					
regionized benchmarks					
cfrac	2.30 ± 0.00	5.22 ± 0.01	2.27×	4.55 ± 0.00	1.98×
mini_httpd-1.15c	2.05 ± 0.00	2.09 ± 0.00	1.02×	2.08 ± 0.00	1.01×

Table 4: Benchmark performance

surprise, since at least one (`grobner`) dates back to the mid 1980s. On the other hand, this is consistent with research that shows that such bugs can linger for years even in widely used software [28].

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 could result in access to `file` being denied (if the region of memory just before the string constant `".htpasswd"` happens to contain that file name), or it could cause the program to crash (if the region of memory is inaccessible). Cyclone found the error with a run-time bounds check.

The `grobner` benchmark had a more serious violation affecting both safety and correctness. The program 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 unpredictable. Four of the ten test cases provided in the distribution follow this code path (in our performance experiments above, we consider only the six correct input cases). In Cyclone, our bounds checks quickly illuminated the source of the problem.

The `tile` program had array bounds violations due to an off-by-one error and an order-of-evaluation bug in this code:

```
if (snum > cur_sentsize)
  mksentarrays(cur_sentsize,
               cur_sentsize += GROWSENT);
```

The function `mksentarrays` reallocates several global arrays. Reallocation is supposed to occur when `snum` is greater than *or equal to* `cur_sentsize`; the `if` guard above has an off-by-one error. Cyclone caught this with a bounds check in `mksentarrays`. In addition, the first argument of `mksentarrays` should be the old size of the array, and the second argument should be the new size. Our platform uses right-to-left evaluation, so the code above passes the new size of the array to `mksentarrays` in *both* argu-

ments. Again, this was caught with a bounds check in Cyclone. In C, the out-of-bounds access was not caught, causing an incorrect initialization of the new arrays.

4 Design history

Cyclone began as an offshoot of the Typed Assembly Language (TAL) project [30, 20]. 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 used only 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 programmers in safe languages like Java — 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 [29], 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 enumerations, since an enumeration 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 enumeration 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 an enumeration variable. 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 supports only 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 assignments to fat pointers are not atomic, and hence, are not thread-safe. We plan to address this by requiring the programmer to acquire a lock before accessing a thread-shared fat pointer; this will be enforced by an extension of the type system. Locks will not be necessary for thread-local fat pointers.

We are experimenting with a number of new pointer representations. For instance, a pointer to a zero-terminated array can be safely represented as just an address, as long as the pointer only moves forward inside the array, and the zero terminator is never overridden. The new representations should make it easier to interface to legacy C code as well as improve on the space overhead of fat pointers.

Finally, we plan to explore ways to automatically translate C programs into Cyclone. We have the beginnings of this in the compiler itself (which tries to report informative errors at places where code needs to be modified), and in a tool we built to semi-automatically construct a Cyclone interface to C libraries.

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. Some other examples are LCLint [17, 24], Metal [13, 14], SLAM [3, 2], PREfix [5], and cqual [32].
2. Inserting run-time checks. C's `assert` statements, the Safe-C system [1], 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 [9, 4, 19] and `printf` format string attacks [8].
3. Combining static analysis and run-time checks. Systems like CCured [31] perform static analyses to check source code for safety, and automatically insert run-time 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 run-time 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. All of these decisions were made because CCured is most concerned with porting legacy code with little or no change; Cyclone is concerned with preserving C's hallmark control over low-level details such as data representation and memory management, both when porting old code and writing new code.

7 Conclusion

Cyclone is a C dialect that prevents safety violations in programs using a combination of static analyses and inserted run-time 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.

Acknowledgements

This research was supported in part by NSF grant 9875536; Sloan grant BR–3734; AFOSR grants F49620–00–1–0198, F49620–01–1–0298, F49620–00–1–0209, and F49620–01–1–0312; ONR grant N00014–01–1–0968; and NSF Graduate Fellowships. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] CERT. Double free bug in zlib compression library. Advisory CA–2002–07, March 12, 2002. `http://www.cert.org/advisories/CA-2002-07.html`.

[8] 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.

[9] 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.

[10] Cyclone. `http://www.cs.cornell.edu/projects/cyclone/`.

[11] 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.

[12] 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`.

[13] 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.

[14] 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.

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

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

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

[18] 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.

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

[20] Dan Grossman and Greg Morrisett. Scalable certification for typed assembly language. In *3rd International Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 117–145, Montreal, Canada, September 2000. Springer-Verlag.

[21] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. ACM, June 2002.

[22] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

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

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

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

[26] Greg McGary. Bounds checking projects. `http://www.gnu.org/software/gcc/projects/bp/main.html`.

[27] MediaNet. `http://www.cs.cornell.edu/people/mhicks/medianet.htm`.

[28] 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.

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

[30] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *Second ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, 1999. Published as INRIA Technical Report 0288, March, 1999.

[31] 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.

[32] 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.

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

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