Safe Systems Programming Languages

Peng Li
Department of Computer and Information Science
University of Pennsylvania

October 6, 2004

Abstract

The C programming language provides explicit memory management, precise control over low-level data representations and high code efficiency. These features are indispensable for systems programming. However, C achieved these goals at the cost of sacrificing type safety. Safety violations like array out-of-bound accesses and dangling pointer accesses lead to a huge amount of well-known software bugs and malicious attacks.

This paper surveys languages and tools for type-safe systems programming, including SafeC, CCured, Vault and Cyclone. Techniques such as code transformation, dynamic safety checking, static analysis, region-based memory management and linear type systems are used to provide type safety for C-like languages. We motivate the use of such languages and tools, present their approaches with examples and explanations, compare them in various aspects and discuss their trade-offs. We present some similarities and dualities among these langauges and provide thoughts about future research directions.

Contents

1 Introduction

1.1 Systems programming languages

We are interested in programming languages for building large-scale, infrastructural software systems such as operating systems, device drivers, databases and network servers. These tasks pose several demanding requirements to the design of the programming language:

- Performance: The program should run as fast as highly optimized hand-written assembly code. In some demanding situations, a 50% performance drop is just unacceptable, no matter whatever benefits the language provides.
- Explicit memory management: At the language level, the most important type of machine resource is the memory. Memory resources are usually limited; the programmer must have the ability to manually (and efficiently) allocate and deallocate memory.
- Fine-grained control over machine-level data representation: Programming with hardware interfaces and network protocols usually requires control over bit-level details of data structures.

1.2 What is wrong with C — safety

In the 1970s, these functional requirements naturally made the C programming language the *de-facto* choice for system programming. Currently, most existing software systems are written in C. However, nowadays people have already realized that C lacks a crucial feature as a language for building secure and reliable infrastructural software systems:

- Safety: *All* program errors must either be detected before execution or be gracefully handled during execution; such errors shall not lead to unpredictable system behaviors.

The safety of C programs becomes more problematic when computers are connected to the internet, where *security* is a big concern. Numerous low-level safety violations such as buffer overflows, out-of-bound array accesses and dangling pointers accesses of software systems have been exploited by hackers to subvert systems and gain unauthorized control. In some worst cases, a hacker could simply send a carefully chosen string to a web server and executed any code on that server, for example, formatting the hard disk or sending fake email messages.

Since the 1990s, much effort has been spent on fighting against software bugs and making patches to existing systems. It is true that, even with a unsafe language like C, software can be made extremely reliable by using mature software engineering techniques and well trained developers. However, such methods cannot achieve an absolute degree of safety: they can get rid of most of the

bugs that are relatively easy to find, but they cannot guarantee that there are no safety violations in millions of lines of code, because the underlying programming language, C, is not type-safe or memory-safe. Some safety violations are extremely hard to detect and reproduce: it may take years for the developers to find them and fix them, and if the attacker finds such a bug before the developers do, an deadly attack may be on its way.

1.3 The need for safer programming tools

It is well known that improving security is an engineering process and it is hard to define and achieve absolute security. However, *safety* is a much simpler property to be enforced at the language level. By using a type-safe language such as Java or OCaml, most safety-related security violations can be avoided: type safety guarantees that safety violations must either be detected before execution or be gracefully handled during execution; such violations will not lead to unpredictable system behaviors. That is to say, a type-safe language can achieve an absolute degree of safety. Therefore, although security breaches such as denial-of-service attacks are still possible, we can guarantee that certain bad things, such as arbitrary code execution caused by memory corruption, will definitely never happen.

Nevertheless, most existing type-safe languages are not the right tools for systems programming, because they do not satisfy the functional requirements listed in Section 1.1. Take Java and OCaml for example: their performance is hurt by run-time safety checks and garbage collection, they do not have explicit memory memory management, and they do not give programmers control over machine-level data structures. Safety does not come for free: it has a price. Systems developers are still using C today because the price of safety is too high: they would take some security risk rather than making their code much slower, eating much more memory and losing control to low-level details.

Therefore, it is necessary to design safe languages particularly for systems programming, with the right trade-off between safety and functionality. Such languages should have explicit control on memory and data layout; they do not need to be high-level and abstract although such features are also desirable. They should provide an absolute degree of safety, and most importantly, the price paid for safety should be reasonably affordable. More specifically, such costs include the following:

- Performance overhead: pointer and array accesses are usually guarded by dynamic checks, such checks hurts performance. Depending on the mechanism used, memory management could also slow things down.
- Memory overhead: the run-time safety checks often rely on additional information stored in built-in data structures such as pointers and memory blocks, causing the program to use more memory than unsafe C programs.
- Type annotations: program analysis tools and advanced type systems often require additional type annotations to be explicitly written down

by the programmer in order to gain necessary static knowledge about the program. Such type annotation can be tedious to write and difficult to read.

- Effort of porting legacy C code: for a language different from C, the difficulty of porting existing C programs to the new language depends on its similarity to C in various aspects including syntax, grammar, common idioms, code organization, data representation, etc.

1.4 Overview

We have motivated the development of safe systems programming languages. The rest of the paper gives a survey of important works in this field, compares them in various aspects, and attempts to identify further research directions.

This paper focus on languages and tools that guarantees safety. There have been a lot of research projects on improving systems software quality, but we mainly study those with absolute safety guarantees. Non-exhaustive bug-finding tools are not in the scope of our survey, although they are closely related and a lot of technologies are similar.

We present the research topics in two major categories. The first category includes SafeC [3] and CCured [40], which mainly focus on compatibility: how to make existing legacy C code safe without completely rewriting them. The second category discusses Cyclone [1] [23] [30] [28] and Vault [19] [13], which are safe alternatives to C with novel language constructs and idioms specifically designed for systems programming.

2 A Taxonomy of Safety Violations

One of the most powerful features of the C programming language is the freedom of using pointers: the programmer has explicit and unrestricted control over the machine memory. At the same time, pointers are the root of safety violations of C programs: the programmer can perform arithmetic operations and type casts on pointers and manipulate them in arbitrary ways, making them point to any places in the memory. Array accesses are similar to pointer accesses: the array bounds are not checked at run-time. Faulty pointer and array accesses can lead to unpredictable behaviors of the system. As summarized by Austin et al. [3], there are two kinds of *memory access errors*:

- *Spatial access error:* The dereference of a pointer or a subscripted array reference is outside the object to be referred. For example, array out-of-bound errors belong to this kind.

- *Temporal access error:* The dereference of a pointer is outside the lifetime of the referent. For example, accessing a heap object after it has been freed, or using a pointer referring to a children stack frame that has been popped out of the stack.

It is worth pointing out that safe dereferencing of pointers can be in some sense treated as the dual problem of safe memory deallocation: when a pointer is dereferenced, we have to make sure that the object is alive, i.e. it has not been deallocated; when we deallocate an object, we have to make sure that no subsequent accesses will be made to it.

In addition to memory errors caused by the use of pointers, there are other kinds of common safety violations in C:

- *Type cast error:* The type of a pointer can be cast to an arbitrary pointer type so that the programmer can easily treat a piece of memory as if it has a different type. Union types have similar effects. Furthermore, pointers and integers can be converted to each other. These powerful features makes it easy to control the low-level details of the machine at the cost of destroying type safety. Misuses of these features can lead to catastrophic effects.

- *Memory leak error:* Some faulty programs fail to release all the memory they allocate; the memory usage of such programs accumulates over time. Such a program will eventually crash when it runs out of memory. Memory leak errors are sometimes hard to find — even automatic garbage collection does not always relieve programmers from such nightmares.

Most safety violations, except memory leak errors, are gracefully handled in a type-safe language. Take Java as an example. For spatial errors: the programmer cannot perform arithmetic operations on pointers; the type casts are dynamically checked against the typing information of the run-time object; array subscripts are dynamically checked against the size of the array. For temporal errors: all the objects except local, stack-allocated variables of base types are allocated in the heap and managed by the garbage collector. For type-cast errors: unsafe casts are dynamically checked during run-time to guarantee type safety. Similar goals should be achieved by a safe systems programming language.

3 Legacy Code Safety

3.1 SafeC: dynamic checking via compiler transformations

This subsection presents SafeC [3], a source-to-source translator for the C language designed by Austin et al. SafeC requires little change to the source code. The basic idea is to change the representation of pointers to an extended structure that contains safety information such as memory bounds. The compiler transforms conventional C programs by converting pointer representations and inserting necessary run-time checks.

```
typedef{   <type>     *value;
           <type>     *base;
           unsigned    size;
           enum {Heap=0, Local, Global} storageClass;
           int capability;   /* plus FOREVER and NEVER */
} SafePtr<type>
```

Figure 1: Safe Pointer Definition

3.1.1 Pointer representation

The definition of a safe pointer representation is shown in Figure 1. A safe pointer contains the value of the pointer as well as other information called *object attributes*, which describe the geometry information of the object referred to by the pointer. Such attributes include:

- `base` and `size`: the base address and the size of the memory object.
- `storageClass`: the memory region that the object lives in. It can be either `Heap, Local` or `Global`. This field helps identify the location of objects and prevent absurd deallocations.
- `capability`: a *unique* identifier for dynamically allocated memory blocks (objects). Each stack frame is given a unique capability at the time it is created; each call to `malloc()` also returns a pointer structure with a unique capability.

Spatial memory access errors are handled easily by checking the pointers against the `base` and the `size` fields of pointer structures at run-time. It is more expensive to handle temporal memory errors. The run-time library maintains a database of capabilities by intercepting the following events:

- Allocation: a unique capability is created and added to the database when a memory block is allocated by entering a stack frame or calling `malloc()`. The returned pointer will have this capability.
- Deallocation: the capability is removed from the database when its corresponding memory block is deallocated by leaving a stack frame or calling `free()`. If the capability is not in the database, a spatial error is detected.
- Memory access: prior to dereferencing each pointer, a run-time check is inserted to check its corresponding capability. If the capability is not in the database, a spatial error is detected because the referent must have been freed earlier. The use of dangling pointers is thus detected.

Thus, the capability database keeps track of all dynamically allocated storage at run-time; temporal errors can be detected by dynamic checking using such information.

3.1.2 Program transformations

The compiler transformation includes three stages: pointer conversion, check insertion and operator conversion. In the first stage, pointer definitions and declarations must be converted to the extended structure that include object attributes. Since pointers are first-class values in C, such transformation must preserve the call-by-value semantics: whenever a pointer is copied in the source program, the pointer structure must be copied as a whole in the target program. In the second stage, dynamic checks are inserted prior to every pointer access. Finally, operations on pointers, especially the '`&`' operator, require special handling. Run-time support is easily added by overloading the memory management library calls and by instrumenting code around function call and returns.

3.1.3 Evaluation

The most significant advantage of SafeC is that it requires little modification to the C source code: legacy code can be made safe without modifications or extensive annotations. On the other hand, it provides seemingly complete safety coverage on spatial and temporal memory errors.

It is also easy to see that SafeC achieved these goals at extremely expensive costs. The performance overhead, as claimed by the authors, range from 130% to 540%, due to the exhaustive run-time checks. It is quite possible that the worst-case performance overhead could be much higher than claimed. Each pointer access is guarded by comparisons against the memory boundaries as well as a query to the capability database. Although the database queries can be efficiently engineered, such overheads are usually much more expensive than the useful computation itself. Treating the pointer structures as first-class values also increases the spatial overhead, as each pointer now becomes four times larger.

However, even such high prices do not bring us a decent safety guarantee: there is no formal soundness of SafeC. In fact, SafeC is unsound in many ways. The object attributes in the pointer representation used by SafeC provide enough information required for run-time safety checks, but the integrity of such objects attributes is not properly protected at run-time. For example, a union type can be used to modify the representation of pointers in arbitrary ways. Similarly, casting between pointer types is also problematic. Fortunately, most programmers do not often intensionally tamper with the representation of pointers. In some worst cases programmers may need to cast from constants or integer types to pointers, such casts can be implemented by API calls that specify appropriate object attributes for the extended pointer structure. Overall, SafeC provides limited safety guarantees with heavy overheads. Such characteristics makes SafeC only useful for debugging programs.

It is interesting that, as run-time mechanisms are added to enforce safety checks, the integrity of the mechanisms themselves can become problematic: a systems programmer should have the ability to access low-level details of

the machine, but should not be able to tamper with the safety enforcement mechanism. These two goals often contradict each other, and sometimes trade-offs have to be made in practical systems.

3.2 CCured: static analysis and type inference

Run-time checking can make existing C programs type-safe but it also causes significant performance overheads. The idea of the CCured system [40] is based on the fact that, in many C programs, most pointer operations are safely used just as if they were written in a type-safe language. Many pointer operations can be statically verified to be type safe and many run-time checks are not necessary. In the CCured system, safe pointer types are distinguished from unsafe pointer types; the type system statically enforces proper operations on pointer types and generate less dynamic checks for safe pointers types.

	C	CCured
Untyped Pointers	`<type> *`	`<type> ref DYNAMIC`
Typed Pointers	`<type> *`	`<type> ref SAFE`
Typed Sequence Pointers	`<type> *`	`<type> ref SEQ`

Figure 2: CCured Pointer Types

3.2.1 Safe pointer types

In the CCured type system, there are two kinds of pointer types: untyped and typed. Untyped pointer types are annotated with `DYNAMIC`. At run-time, the representation of untyped pointers include the bound information of the target memory block and all pointer accesses are guarded by dynamic checks. Any data structures referred to by untyped pointers is also untyped. As for typed pointers, there are two types: safe pointers and sequence pointers, annotated by `SAFE` and `SEQ` respectively. The type system guarantees that a non-null safe pointer always point to a well-typed structure. The only necessary run-time check on safe pointers is the null-check, which can be efficiently implemented. A sequence pointer can be understood as a pointer to a well-typed array: pointer arithmetic can be performed on sequence pointers. Any element in the array is well-typed. Sequence pointers have boundary information in their run-time representations just like untyped pointers, but the type system guarantees that sequence pointers point to well-typed structures as long as array out-of-bound errors do not happen.

An example of using these pointers is shown in Figure 3. The CCured type system provide the following guarantees: On line 5, the sequence pointer `p` points to a well-typed array: each element in the array is a structure of two well-formed pointers. On line 9, the safe pointer `e` points to a well-typed structure, because sequence pointers can be cast to safe pointers after a bound check. Accesses to `e` require no dynamic checks as the type system guarantees that it points

```
01  struct elem {
02     int ref DYNAMIC unsafe_pointer;
03     int ref SAFE safe_pointer;
04  }
05  int func(struct elem ref SEQ p) {
06      struct elem ref SAFE e = NULL;
07      int s = 0;
08      for (int i=0; i<100; i++) {
09         e = (struct elem ref SAFE) (p+i);\\ Out-of-bound check
10         s += *(e->unsafe_pointer);       \\ Out-of-bound check
11         s += *(e->safe_pointer);         \\ Only NULL-check
12      }
13      return s;
14  }
```

Figure 3: Safe and unsafe pointers

to a valid location. On line 10, a out-of-bound dynamic check is required for `unsafe_pointer`. On line 11, no out-of-bound check is required because the typing information guarantees that `safe_pointer` is either null or points to a safe memory location.

The soundness of CCured is achieved by making barriers between the typed world and the untyped world in the type system. All values (pointers) and memory blocks are either typed or untyped. Typed pointers only point to typed memory blocks; untyped pointers only point to untyped memory blocks. More specifically, the following invariant holds:

- Typed pointers cannot be cast to untyped pointers, and vice versa. Integers can only be cast to untyped pointers, but run-time coercions are inserted to make sure that the boundary information on such pointers are always vacuous, i.e. accesses to such pointers are forbidden.

- Typed memory area can contain untyped pointers, but untyped memory area cannot contain typed pointers. That is to say, all the pointers in the untyped memory area are also untyped. The only possible way to link typed and untyped worlds is when a typed memory area contain a untyped pointer to untyped memory.

- In the untyped world, i.e. memory areas referred to by `DYNAMIC` pointers, a tag is maintained for each word to distinguish pointers from integers in the memory. As a result, each untyped memory area is associated with a bitmap of tags. When a untyped pointer is read from a untyped memory area, the corresponding tag is checked to ensure the integrity of pointers. These tags also make garbage collection efficient.

The innovation of CCured mainly focuses on handling spatial memory access errors. For temporal memory access errors, CCured simply forbids explicit deallocation and uses the Boehm-Weiser conservative garbage collector [7] [8], which periodically scans through the memory and searches for pointer references to memory blocks. To capture temporal errors in the stack, some conservative restrictions and dynamic checks are used.

3.2.2 Type inference

The static analysis of CCured heavily depends on the typing annotations of pointers. Such annotations can be explicitly written by the programmer using the `__attribute__` keyword of GCC, but it would be a lot of work to annotate all the pointer declarations. CCured has a type inference algorithm that takes a C program and constructs a set of pointer-kind qualifiers that makes the program well-typed in CCured. The algorithm introduces a qualifier variable for each pointer declaration, generates a set of typing constraints among these variables during typechecking, and attempts to solve these constraints. Apparently, the most conservative solution is to make all pointers `DYNAMIC`. The inference algorithm use some strategies that attempts to maximize the number of `SAFE` and `SEQ` pointers.

3.2.3 Evaluation

Compared to purely dynamic approaches, the static typing analysis of CCured greatly reduces the overhead of run-time safety checks. In the benchmarks used, the inference algorithm detected that most pointers are either `SEQ` or `SAFE`. Most of the benchmarks exhibited overhead between 30% to 150% compared to original performance. Compared to SafeC, CCured also benefits from its simpler run-time pointer representation: the only additional information added to the pointers is the memory bound. The CCured type inference algorithm is designed to minimize the amount of source changes. Manual editing is still necessary in some situations such as low-level type casts and data structures affected by the size of pointers.

While in SafeC the representation of pointers may be subverted by type casts and aliasing, CCured is completely type-safe in this aspect. The most significant trade-off is performance: a bitmap of dynamic typing information is maintained for each untyped memory area; accesses to each word is dynamically checked and recorded. This problem is alleviated by the fact that most pointer and memory areas are statically typed. Other trade-offs for soundness are flexibility and compatibility: type-safe wrapper interfaces can be written for some existing libraries including the standard C library, but some third-party libraries are difficult to intergrate with the CCured type system, especially those containing pointers in the data structures.

Perhaps the most inconvenient limitation of CCured is that it does not give programmers the flexibility on memory management: it relies solely on a garbage collector. Although conservative garbage collectors are well-engineered

and practically usable in many applications, the lack of explicit memory deallocation is unacceptable in certain areas of systems programming.

3.3 Related work

As implemented in SafeC and CCured, the most effective approach to achieve spatial memory safety seems to be the use of fat pointers that include additional bounds information in their run-time representations. Both Steffen's safe compiler RTCC [50] and Kendall's C source translator Bcc [33] focus on such spatial error checking. To avoid the incompatibility caused by the change of pointer representation, Jones and Kelly [31] store extra information for run-time checks in a tree structure, allowing pointer representations to be backward compatible at a slowdown factor 5 or 6. To reduce the performance overhead, Fischer and Patil presented a system [44] that uses additional processors to perform the bound checks.

Purify [25] is a memory access checking tool for binary object code, where spatial safety is treated in a much coarse-grained form: memory accesses are only checked against the bounds of whole heap area. This is similar to the goal of software fault isolation [55].

SafeC uses a complex run-time mechanism to capture temporal memory errors. There have been some similar but less reliable "smart pointer" implementations [14] [22]. Saber-C [32] use interpreters to capture temporal memory errors. On the other side, conservative garbage collectors [8] [7] are used in CCured as well as in many other systems including Purify [25].

Whereas CCured uses memory bitmaps to store dynamic typing information for pointers, Loginov et al. [36] store more fine-grained dynamic typing information for each memory location at much heavier costs. CCured also uses static analysis to reduce the dynamic checking overheads. Much work [26] [29] [34] [46] [10] [59] has been done to remove dynamic checks from dynamically typed languages like LISP, but less work has been done for languages like C. Ellis and Detlefs defined a safe subset of C++ [15]. Chandra et al. [11] and Siff et al. [47] presented methods for checking casts between structure types. There have been a lot of unsound yet useful static analysis tools for finding memory bugs in C code, such as LCLint [18] [35] and PREFix [9].

4 Safe Variants of C

4.1 Cyclone: region-based memory management

4.1.1 Overview

Cyclone [1] [23] [30] [28] is a safe programming language that looks very similar to C. Cyclone inherits most of the lexical conventions and grammar from C, but it imposes necessary restrictions to preserve type safety. For example, unsafe type casts are disallowed, pointer arithmetic is restricted, `NULL` checks are

required when pointers are used. To overcome these restrictions and regain the programming idioms of C, Cyclone provides a lot of novel language extensions.

Pointers are treated in a well-typed fashion in Cyclone. Besides the conventional C pointer types like "`FILE * a`", two finer types for pointers are introduced. A *non-null pointer* can be defined as "`FILE @ b`", meaning that this pointer will never be `NULL`. Functions that work with this pointer type do not need run-time `NULL`-checks, so both safety and efficiency are achieved together. Run-time `NULL` checks are only inserted when a conventional pointer is cast to a non-null pointer. A *fat pointer* is a pointer that supports pointer arithmetic. Instead of writing "`int *s`" in C, we write "`int ?s`" in Cyclone to define a fat pointer `s`. Fat pointers have bounds information in their run-time representations and accesses are guarded by dynamic checks. Furthermore, Cyclone uses static analysis to rule out the use of uninitialized pointers: all pointers must point to meaningful locations before they are used (dereferenced, passed to function calls, etc).

Unions and type casts also have their type-safe counterparts. Cyclone supports *tagged unions* similar to ML data types. A tagged union has a special tag in its run-time representation; the value of the tag uniquely decides which branch is taken. Values of tagged union types can be pattern-matched by using an extension of the `switch` statement. Many uses of "`void *`" in library interfaces can be replaced by using polymorphic types.

Cyclone provides a wide variety of memory-management options. The core feature of Cyclone is *region-based memory management*, a type-based mechanism for enforcing spatial memory safety. Various kinds of regions are implemented for different purposes; the detail is illustrated in the following sections.

4.1.2 Region-based memory management

A region can be thought of as a segment of memory. The idea of region-based memory management is very simple: each object is allocated in some region; a region's objects are all deallocated simultaneously. For example, a stack frame can be thought of as a region: when the stack frame is popped off the stack, all local variables are deallocated at once. This mechanism is based on the observation that in many situations we require a group of objects be deallocated at the same time rather than deallocating them individually at different time.

Region-based memory management has many benefits. The programmer has explicit control of the location and lifetime of memory objects: she can choose the region in which an object lives; she can also choose the time to deallocate the region. It is more efficient than deallocating objects individually or using garbage collectors. Memory leaking nightmares are also less likely to happen.

In Cyclone, regions are built into the type system. Each pointer points into exactly one region. For example, the pointer `p` declared as "`int *‘r p`" points to the region `r`, where `r` is the name of the region in the type system. Each lexical block corresponds to a region; the region can be named by the label of the block. Figure 4 shows an example of using region types. On line 2, the pointer `p` has type `int ‘L1` because it is a pointer to region `L1`, where `L1` is the

```
1.  L1: { int a = 3;
2.        int *‘L1 p = &a;       // safe
3.        L2: { int b = 4;
4.              int *‘L2 q = &b; // safe
5.              q = &a;          // safe, L1 outlives L2
6.              p = &b;          // type error!
7.              bar( p, q ); // function call
8.        }
9.        *p = 5; //potential dangling pointer
0.  }
```

Figure 4: Regions in Cyclone

label of the lexical block. The Cyclone type system rejects potential dangling pointer accesses. On line 6, `p` tries to point to the region `L2`. This is rejected by the compiler due to a type mismatch because `&b` has type `int *‘L2` and `p` has type `int *‘L1`. This avoids a potential memory error: when we leave the lexical scope of `L2` on line 8, assuming the stack region of `L2` is deallocated, `p` would become a dangling pointer on line 9.

However, the pointer assignment on line 5 is not rejected by Cyclone. This is based on the observation that, whenever region `L2` is alive, `L1` is always alive, i.e. `L1` *outlives* `L2`. Therefore, it is safe to let q point to region `L1` because doing so will not create the possibility of having dangling pointers. In the Cyclone type system, this is achieved by defining a subtyping relation on pointers based on the "outlives" relation on regions: if `r1` outlives `r2`, then `int *'r1` is a subtype of `int *'r2`.

On line 7 of Figure 4, there is a function call to `bar(p,q)`. How can we write the function declaration for `bar`? We cannot write `L1` and `L2` because they are not in the lexical scope when we define a function; doing so also limits the use of the function to other regions. Functions in Cyclone are *region-polymorphic*; they use region variables to abstract over the actual regions of their arguments or results. Figure 5 shows a possible implementation.

```
1.  int *‘r1 bar<‘r1,‘r2>( int *‘r1 x, int *‘r2 y) {
2.      *x = *y + 4;
3.      return x;
4.  }
```

Figure 5: Polymorphic region parameters

The parameters' region names are abstracted by the syntax `<‘r1,‘r2>` . When we use the function, `r1` and `r2` are instantiated with `L1` and `L2`, respectively.

In addition to stack regions, Cyclone also provide *growable regions* that allocate regions in the heap during run-time. On the left side of Figure 6, the growable regions are lexically scoped: the declaration on line 2 creates a grow-

able region. The region type is `r` and it corresponds to a run-time region handle `h`. On line 3, memory is allocated on region `r` by calling a library function using handle `h`. On line 4, more memory can be allocated on that region during some computation. When the execution leaves the lexical scope of region `r` on line 5, the growable region is automatically freed.

```
   // Lexical Growable Region:

1. void foo() {
2.  { region<`r> h;
3.    int *`r x=rmalloc(h,100);
4.    allocate_more_on(h);
5.  }  //auto deallocation
6. }
7.
```

```
   // Dynamic Growable Region:

1.  void foo(dynregion_t<`r, `H> k) {
2.   { region h = open(k); // check
3.     int *`r x = rmalloc(h, 100);
4.     allocate_more_on(h);
5.   }
6.   free_dynregion(k);
7.  }
```

Figure 6: Growable Regions

So far, all the above lexical regions always follow a LIFO order, which makes static checking simple and naturally provides subtyping on regions. However, LIFO has its limitations: an object's deallocation time is fixed once it is allocated. Dynamic regions are not restricted by the LIFO discipline. The right side of Figure 6 shows an example of *dynamic region*: `k` is the "key" of the region, which dynamically records the status of the region. To access a dynamic region, the programmer has to "open" the region using a library call. The run-time library will use the key to verify that the region is indeed alive, and binds the handle to `h`. If the region has been freed, an exception will be raised. The keys are very similar to the capabilities in SafeC [3]. Inside the scope of `h`, the region can be used just like a lexically growable region. At the end of the scope on line 5, instead of being deallocated, the region is simply detached from the handle `h`. The use of this region is therefore not limited to this function: other functions can also use it by opening the key `k`. Deallocation requires an explicit call to `free_dynregion(k)`, which verifies the key status and raises an exception if the region has already been freed.

Finally, Cyclone provides a special *heap region* where an optional garbage collector can be used to deal with hopeless cases that region-based memory management cannot handle.

4.1.3 Unique pointers and linearity

In addition to region-based memory management, *unique pointers* are another way to achieve safe and explicit memory deallocation. The idea is that if a pointer has no aliases in the system, it is safe to stop using this pointer and deallocate the memory it points to. In Cyclone, a unique pointer can be declared in the syntax of `int *`U p`, meaning that the pointer `p` is unique. The letter `U` is distinguished from region names. Figure 7 shows an example of using unique pointers. The Cyclone type system prevents unique pointers from being aliased:

```
1. struct pointer { int x; int y; } *‘U p;
2. p = malloc (sizeof(struct point));
3. p -> x = 1;
4. p -> y = 2;
   ......
5. free(p);
```

Figure 7: Safe deallocation of unique pointers

we cannot create copies of such pointers. A call to `free` on line 5 *consumes* the unique pointer: such a pointer cannot be used after it has been consumed. To prevent memory leaking, the type system can statically enforce that all unique pointers must be consumed at the end of its scope.

```
1. int *‘U g = NULL;
2. void init(int x) {
3.   int *‘U temp = malloc(1000);
4.   *temp = x;
5.   g :=: temp;
6.   if (temp!=NULL) free(temp);
7. }
```

```
1. int *‘U p = malloc(sizeof(int));
2. *p = 3;
3. { alias <‘r2> int *‘r2 tmp = p;
4.   int *‘r2 q = tmp;
5.   int *‘r2 r = tmp;
6.   *tmp = (*q) + (*r);
7. }
8. free(p);
```

Figure 8: Swap and Alias

In order to make the type system sound, there are strong restrictions on unique pointers. It is particularly annoying that any shared data structures are inherently aliased: a global variable can be accessed from any place in the program. In practice, it would be too restrictive to forbid sharing of unique pointers. Cyclone solves this problem by introducing the *swap* operation: shared unique pointers cannot be directly accessed; they have to be used by swapping them with other unique pointers. The left side of Figure 8 shows an example of using globally shared unique pointers and swapping. On line 5, the global unique pointer is updated by using the swap operator "`:=:`". This way of handling unique pointers is safe because swapping does not introduce new copies of pointers. At any time, there is at most one reference to the memory location pointed by unique pointers.

Cyclone also support *temporary aliasing* of a unique pointer in a lexical scope. The key idea is to prevent aliases from being leaked to the outside of the aliased scope. The right side of Figure 8 shows an example of temporary aliasing. Although `p` is a unique pointer, it can be aliased in the block on line 3-7. A fresh region name `‘r2` ensures that no pointers to the aliased memory will escape from its lexical scope. On line 8, `p` is a unique pointer again and it can be safely deallocated. This is a very clever use of the region type system in Cyclone: aliasing can be implemented using just lexical regions.

4.1.4 Evaluation

Overall, Cyclone is a safe dialect of C. It is carefully designed with the goal of safety. While accommodating the low-level programming style like C, it provides the safety of a high-level programming language like Java. Cyclone has an advanced type system. Although region-based programming in C is not a new idea, Cyclone fully integrates regions into its language design and offer static type safety guarantees. The variety of memory management options makes Cyclone very flexible for different tasks.

The performance of Cyclone is quite good. In some rare situations, programs written in Cyclone can even have better performance than in C because the region-based memory management is more efficient. Compared to C, the overhead generally varies from 0% to 150%, mostly depends on the pattern of the computation performed and the memory usage. The major overheads due to safety are fat pointers and garbage collection. These problems seem unavoidable, but they have been largely alleviated by Cyclone's extensive static checking and the region-based memory management.

Porting C programs to Cyclone could take a lot of work. Although Cyclone looks similar to C, it does require the program to be completely rewritten. Most well-written C programs can be ported to Cyclone with merely syntactic changes and work well with the garbage collector, but they require more significant changes to fully utilize the region-based memory management provided by Cyclone. The Cyclone compiler can infer many region type annotations, but the programmers have to understand the type system and write down type annotations when needed. On average, one manual region annotation is needed for about 150 lines of code. Writing library code usually requires much more type annotations to make the interfaces flexible and reusable in the Cyclone type system.

4.2 Vault: practical linear type system

4.2.1 Overview

Vault [19] [13] is a safe programming language developed by Microsoft Research. Although it is intended to be a safe version of C or C++, many of its features are actually closer to Java or C#. Memory objects such as `struct` are indirect by default; they have to be accessed through pointers. The C-style pointer arithmetic is not supported. Vault also provides many advanced language elements such as built-in string types, modules, closures, variants and generics.

The most novel feature of Vault is its type system. Vault supports linear types to track object usages at compile-time. Furthermore, the linear type system also allows state be attached to object types. Such state can be used to statically enforce resource management protocols on the tracked[1] objects.

[1] In this paper, "unique", "linear" and "tracked" have the same meaning when talking about pointers and objects.

4.2.2 The linear type system

In the Vault type system, resources are represented by *keys*. A key is a unique name for some resource. Keys are also called *capabilities*[2] in similar type systems. For each point in the program, a list of keys can be computed, called the *held-key set*, representing the available resources at that program point.

```
1. tracked(K) point p = new tracked point{x=3; y=4};
2. tracked point q = p;
3. free(p);
4. free(q); // rejected by the type checker
```

Figure 9: Tracked types

A *tracked* type is a linear type in Vault. A tracked pointer is very similar to a unique pointer in Cyclone. Instead of forbidding aliasing of unique pointers, the Vault type system keeps track of all the aliases of a *tracked* type and makes sure that no alias can escape the scope of static analysis. As a result, the type system knows the availability of resources at each program point. This is achieved by using singleton types to distinguish aliases. Figure 9 shows an example. Line 1 creates a new linear object and assign it to the tracked pointer p. The name `K` specifies the key of the newly allocated object. After this line, the held-key set includes `K`, meaning that the resource represented by `K` is available. The pointer has a singleton type `tracked(K) point`, meaning that pointers of this type refer to resource `K`. Any pointer with this type must have the same value of `p`. For example, the type system can give type `tracked(K) point` to `q` on line 2, although the key is omitted in the program. When we free the object on line 3, the type system knows that the object represented by `K` is gone, so that `K` is removed from the held-key set. If we free `q` again, the type system will complain because `q` points to resource `K` but `K` is not available in the held-key set.

So far, the held-key set we introduced only records the *availability* of resources and it is suitable for memory management. In fact, availability can be generalized to represent more information about resources. Since the type system can track all the operations on a given resource, we can assign *states* to resources and record them in the held-key set. For example, the held-key set {`F@closed`} means that, the key `F` is available and the state of that resource is "closed". We can then define functions that change the state of resources, and use the type system to ensure that these functions are used in proper orders. For example, Figure 10 shows an interface specifying protocols on file handles:

A key has state `raw` when the corresponding resource is allocated. On line 1, the additional annotation `[$F@raw->@open]` specifies the pre-condition and the post-condition of this function: the key `F` must be at the state `raw` before calling this function; the state of `F` will become `open` in the held-key set after this call.

[2]The capabilities mentioned here have nothing to do with the capabilities in SafeC.

```
1. void FileOpen(tracked($F) FILE)              [ $F @raw -> @open ];
2. void FileRead(tracked($F) FILE, byte[], int) [ $F @open ];
3. void FileClose(tracked($F) FILE)             [ - $F ];
```

Figure 10: Enforcing protocols on file handles

Furthermore, Vault supports *guarded types* to specify access-control like conditions on types. A type guard is a predicate on the held-key set. For example, the guard `K@open` means that the key `K` must be available in the held-key set and its state must be at "`open`". The predicate must be true at the program point where a variable of the guarded type is accessed. If we change the declaration of a variable from "`FILE input`" to "`K@open:FILE input`", any access to this variable must satisfy the condition that the key `K` is available and at the "`open`" state.

```
1. tracked(R) region rgn = Region.create();   // key R added
2. R:point pt = new(rgn) point {x=1; y=2;};   // guarded by R
3. ....
4. Region.delete(rgn);  // key R is removed
5. pt.x++;              // type error: R is not available
```

Figure 11: Region-based memory management using type guards

Guarded types are useful to specify the condition that the availability of some resource depends on other resources. Particularly, it can be used to safely implement region-based memory management. Figure 11 shows how regions can be used. A region is a linearly tracked, first-class object. Library calls are provided to allocate and free regions as well as allocating memory via the `new` operator. The type system enforces that, the type of objects allocated in a region must be guarded by that region's key. Therefore, the `pt` pointer on line 2 is guarded by `R`, the key of the region allocated on line 1. After the region is freed, the `R` is removed from the held-key set, so that accesses to objects guarded by `R` will become illegal on line 5.

4.2.3 Adoption and focus

The linear type system of Vault is very powerful, but in reality, it is often difficult to use because the type system requires the ability to tell different objects apart when multiple aliases are available. If we cannot tell whether two pointers point to the same object, we cannot enforce protocols on these objects. For instance, the code "`free(a); free(b)`" is safe only if `a` and `b` point to different objects. The type system would be too restrictive if it keeps track of all aliasing information statically, so Vault has both tracked and untracked objects in the language. However, this prevents tracked objects from being shared in some useful ways and also disallows enforcing protocols on untracked objects.

Vault introduces two novel language constructs, *adoption* and *focus*, to make the linear type system more usable: *adoption* allows a tracked linear object be safely shared; *focus* allows shared nonlinear objects be temporarily treated as linear objects.

```
                                        Held-key set after statement
1. tracked(A) ptr p1 = new ...          {A}
2. tracked(B) ptr p2 = new ...          {A,B}
3. B:ptr q = adopt p1 by p2;            {B}      (A is consumed)
4. .....                                {B}
5. free(p2); // p1 also freed           {}
6. q.x = 4;  // type error: B is not available!
```

Figure 12: Adoption

Adoption consumes a linear pointer to an object and results in a nonlinear (not tracked) and guarded pointer. The syntax of adoption is `adopt p1 by p2`, which takes a adoptee `p1` and a adopter `p2`. As shown in Figure 12, both `p1` and `p2` are linear references (tracked types). The result of adoption is a nonlinear version of `p1`. Adoption creates a hidden, run-time linear reference from the object `p2` to `p1`. When `p2` is deallocated on line 5, the run-time memory management mechanism crawls via such references and deallocate `p1` automatically. Therefore, `p1` will have the same lifetime as `p2`. To ensure temporal memory safety, the type system must enforce that any accesses to `p1` is within the lifetime of `p2`. This is guaranteed by using type guards. As shown on line 3, the resulting nonlinear reference has a guarded type "`B:ptr`", meaning that any accesses to `p1` requires the key `B` is alive in the held-key set. `B` is removed on line 5 when `p2` is deallocated and the dangling pointer access on line 6 can be statically detected by the type system.

```
                                        Held-key set after statement
1. B:ptr q = ...;                       {B}
2. B:ptr r = q;                         {B}
3. let f = focus q in {                 {K} (K is fresh)
4.    r.x = 3; // type error            {K} (B is not available)
5.    ......;                           {K} (K is still available)
6. }                                    {B}
7. r.x = 3;   // OK                     {B}
```

Figure 13: Focus

Focus takes an adopted nonlinear object and provides a temporary linear view on it. The syntax of focus is "`let f = focus q in e;`", which takes a nonlinear object `q` and treat it as a linear object `f` in expression `e`. Figure 13 shows an example [3] of using focus. Before line 3, `q` and `r` are nonlinear aliases

[3] To make this example more interesting we should use the fact that `f` is a linear type on

to the same object; they are all guarded by the key `B`. On line 3, the type system removes `B` from the current held-key set to forbid the use of aliases of `f`, namely, `q` and `r`, inside the focused block. Therefore, the illegal aliased access on line 4 is ruled out by the type system. A fresh key `K` is created and added to the held-key set, and the focused linear object `f` will have type `tracked(K) ptr` inside the focused block. At the end of block (after line 5), the type system requires that `K` is still available to make sure that all prior aliases are still valid. It then restores the original environment on line 6.

4.2.4 Evaluation

Vault is a very advanced programming language. Besides appealing features such as modules, closures and generics, it provides a powerful linear type system to give programmers control over memory layout and lifetime. The linear type system also allows static checking on the usage of linear objects. Resource management protocols can be specified using state transitions on function interfaces and can be enforced by the type system. That is to say, Vault not only guarantees *safety* but also ensures program *correctness* with regard to some protocol specifications. Temporal memory safety can be treated as a simple protocol in this framework. The Vault type system is also powerful enough to support region-based memory management in a type-safe fashion.

We expect that the performance of Vault is similar to Cyclone. Compared to other type-safe languages, the only additional overhead of using Vault is to maintain the hidden references used by adoption, which can be efficiently implemented. Therefore, programs written in Vault should perform optimally as in other type-safe languages.

The elegance of Vault comes at the cost of losing *compatibility*. Porting a C program to Vault require the whole program to be completely rewritten. The syntax, the grammar, the module system and library calls are totally different from C. Furthermore, many low-level details are also different: C uses null-terminated strings while Vault use its own string representations; all objects including the `struct` are indirectly accessed through pointers; pointer arithmetic is impossible. Thus, it is possible that the data structures have to be redesigned during the porting process. The powerful type system of Vault requires a huge amount of type annotations to be written down by the programmer. Lastly, the *focus* operation in Vault is unsafe in a concurrent, multi-threaded setting.

4.3 Related work

Gay and Aiken developed their RC [20] [21] compiler that supports region-based programming in C. RC mostly uses reference-counted dynamic regions; accesses to regions are guarded by dynamic checks which can be optimized by writing down pointer annotations.

---

line 5. Doing so will involve significantly more complex data structures with multiple levels of indirection; such examples are shown the paper by Fähndrich and DeLine [19].

Tofte and Talpin presented the theoretical foundations [52] [53] for region-based memory management in ML-like languages. Their ML Kit [51] implements SML with LIFO regions (like Cyclone does) using type inference. More recent work on the ML Kit enables garbage collection within regions [24].

To avoid the LIFO principles on region lifetimes, Aiken et al. [2] provide an analysis to free some regions early. Work on linear types [54] [48], alias types [49] [56] and capabilities [12] provide the foundation for the linear treatment of regions. The Capability Calculus [12] and the work by Walker et al. [57] use linear types for safe deallocation of regions at any program point. The `alias` construct in Cyclone is very similar to Wadler's `let!` construct [57] and its variations [43]. Ideas from such work lead to implementations of type-safe garbage collectors within intermediate languages [58] [38]. Vault also uses linear types for regions, but the type system of Vault appears more general. Niss and Henglein [27] also designed a flexible region system for first-order programs.

A lot of static analysis tools such as Metal [16] [17], SLAM [4] [5] and Cqual [45] are designed to discover bugs in C programs using using rules specified by the programmer. Metal and SLAM share a lot of motivation with Vault; both of them can check advanced program properties such as resource protocols.

5 Comparisons

5.1 Language features

5.1.1 Control over low-level machine details

All the languages and tools we presented provide ways to access low-level data representations. SafeC and CCured works for C programs and therefore fully preserves the power of C. Cyclone is a redesigned language, but it supports most low-level programming styles in C, including pointer arithmetics and null-terminated strings. Vault has more features similar to Java and C#: all the structures are indirectly accessed via pointers and allocated in the heap; Vault has its own string representations. Although a Vault programmer has the ability to control "the bits" of data structures such as `struct` and strings, such operations are generally less convenient and less efficient. For example, it is not possible to take the address of some field in the middle of a data structure and pass the reference to a library call. Vault also requires hidden fields in the run-time representation of objects, which makes Vault more abstract than others.

One limitation of these languages is that the programmer cannot touch the internal representation of pointers and other data structures used to protect safety. Although unsafe, casting between integers and pointers is necessary in some situations for efficiency and functionality. SafeC allows unsafe casts and the use of C unions at the cost of sacrificing safety. CCured allows limited casts: integers can be casted to pointers, but such pointers will have erratic bounds information in them and cannot be directly used.

Short summary: SafeC > CCured > Cyclone >> Vault

5.1.2 Memory management options

SafeC provides the same memory management mechanism as C does where heap allocation and deallocation are all manual and explicit. CCured simply makes `free()` a no-op and solely depends on the conservative garbage collector. Vault provides safe and explicit deallocation of linear objects as well as region-based memory management. The powerful type system of Vault uniformly treats regions as linear objects. Cyclone provides many memory management options including lexical regions, dynamic regions, unique pointers, reference-counted objects and garbage collection. Both Cyclone and Vault support relaxations on linear types.

Short summary: Cyclone > Vault >> CCured ≈ SafeC

5.2 Costs

5.2.1 Performance overhead

Compared to other safety checking tools such as Purify [25], the performance overhead of SafeC is low enough for program testing during software development. However, it is still too high for released software: an instrumented program typically runs several times slower.

CCured, Cyclone and Vault all uses relatively lightweight safety mechanisms and the current implementations of CCured and Cyclone perform very well for existing benchmark programs. The treatment of untyped memory in CCured has the potential to cause high overheads, but it seems to be the best trade-off so far for C programs since C is inherently untyped. We have not yet seen any performance measurement of Vault, but theoretically, Vault does not use more expensive safety mechanisms than Cyclone. Other than the untyped aspect of CCured, two major factors contributed to the performance overhead due to safety checking: bounds checks and garbage collection. Pointer accesses are guarded with bounds checks. It is also more expensive to manipulate fat pointers in memory as the associated bounds information affects register allocation and cache performance. The cost of garbage collection varies widely for different programs.

Bounds checks and garbage collection are sources of overhead for most safe languages including C# and Java. There has been a lot of research on eliminating array bound checks using various program analysis and compiler transformations [6] [37]. Proof carrying code [42] [41] [39] and dependent types [62] [61] [60] also look promising. Many of these approaches are applicable to Cyclone and Vault. We believe that these optimization techniques are important for practical application of safe system programming languages. As we design a type-safe language with performance as a priority and extensive program annotations are acceptable, it would be sensible to design programming idioms that are particularly friendly with optimizations for bound checks. Cyclone and Vault provides advanced memory management options that overcomes the problem of garbage collection: proper uses of region-based memory management and linear objects can lead to both safety and optimal performance.

Short summary: Cyclone ≈ Vault > CCured >> SafeC

5.2.2 Memory overhead

The safety mechanisms require run-time data structures to store the bounds and typing information. Fat pointers are used in SafeC, CCured and Cyclone for safe pointer arithmetics. SafeC uses extensive information in the pointer representations as well as a run-time capability store for tracking memory usage. The data size overhead can be 200% to 400%, which is merely practical for programing testing.

CCured uses bitmaps to store dynamic typing information for untyped memory. Each word corresponds to one bit in the bitmap. Depending on the word size on the machine, the memory overhead due to this bitmap is either 1/32 or 1/64, which seems practical and scalable.

The adoption operation in Vault requires some hidden fields in the run-time representation of objects: each object may be linked to a list of it adopted objects. This can be efficiently implemented using linked lists or other data structures, but adding hidden fields to all objects introduces slight memory overheads and makes the language more abstract.

Short summary: Cyclone > Vault > CCured >> SafeC

5.2.3 Type annotation effort

SafeC uses completely dynamic mechanisms and require no additional typing annotation to be written by the programmer. The static analysis of CCured relies on the typing annotations. Programmers can use these annotations explicitly, but its type inference algorithm can infer most of the useful annotations. The programmer can view the output of type inference using a browser utility and make adjustments if necessary. The annotations in CCured are only used for improving performance.

Although most region types in Cyclone can be inferred automatically, the programmer has to write down some of the region types in order to typecheck the program. Writing reusable library interfaces requires much more extensive typing annotations. The type annotations for linear objects in Vault look even more verbose. Nevertheless, Vault enforces high-level protocols using the type system, and many type annotations can be treated as specifications of programs. These benefits may overcome the verbosity of its ASCII syntax.

Short summary: SafeC > CCured > Cyclone > Vault

5.2.4 Code migration effort

Both SafeC and CCured require very few changes to legacy C programs. CCured allows the programmer to insert type annotations in order to optimize the run-time safety checks, which requires additional efforts. Cyclone preserves many programming styles of C so that most C programs can be ported to Cyclone in an almost line-to-line fashion. To fully utilize the region-based memory management options and advanced features such as polymorphism, the program may

require significant change. Migrating a C program to Vault may require considerable effort because C and Vault are drastically different in many aspects. The details are discussed in Section 4.2.4.

Short summary: SafeC > CCured > Cyclone >> Vault

5.3 Safety guarantees

5.3.1 Spatial memory safety

Fat pointers and array bound checks seem to be the standard approaches for enforcing spatial memory safety: all the safe languages including Java and C# use them. Soundness is easily guaranteed; it is more challenging to reduce the performance impact due to dynamic checks.

5.3.2 Temporal memory safety

All the languages and tools we presented provide complete coverages on temporal memory errors. The mechanisms used can be classified into several categories:

- Dynamic checks: SafeC uses a capability store to record the memory layout and use dynamic checks on each pointer access to ensure temporal safety. The dynamic regions and reference-counted objects in Cyclone also use similar approaches to ensure safety.
- Garbage collection: conservative garbage collectors are used in CCured and Cyclone. The author of this paper does not yet know whether Vault uses a garbage collector, but it probably does.
- Region-based memory management in Cyclone and Vault.
- Linear types: Vault has an advanced linear type system which enforces temporal safety as well as other high-level protocols. Cyclone has limited support on unique pointers.

5.3.3 Type-cast safety

One drawback of SafeC is that it loses soundness when certain type casts and union operations are present. CCured safely handles the untyped elements of C by separating typed memory and untyped memory: dynamic typing information is associated with every word of the untyped memory to prevent pointers from being illegally altered. Cyclone and Vault are designed to be type-safe; dangerous type-casts are statically ruled out by the compiler. Instead of using unions which are inherently problematic, Cyclone and Vault use tagged unions (variants) and pattern matching.

5.3.4 Detecting and preventing memory leaks

Memory leaking is much harder to fight against. The capability store in SafeC can be used to detect memory leak when program runs. At any time, the capability store has a global view of the memory usage, which is more accurate than the information provided by conservative garbage collectors.

Cyclone and Vault prevent more memory leaks by providing advanced memory management mechanisms. Linear type systems guarantee that linear objects must be consumed. Region-based memory management tends to reduce the risk of memory leaking because it is easier to keep track of regions rather than memory objects, plus lexical regions are automatically deallocated.

It is worth pointing out that Cyclone does not guarantee that linear objects are always consumed. The Cyclone type system would be too restrictive if it requires a linear object to have the same status at the end of branches. Vault solves this problem by storing keys in the variants and recovering keys in pattern matching.

5.3.5 Soundness

Although there is no formal proof of soundness for SafeC, it seems to have complete coverage on various memory errors, given that the source program has no unsafe type casts and unions. The core language of CCured is formally sound. CCured has a type system for pointers and it enforces type safety by using dynamically typed memory. Cyclone and Vault are designed from the beginning to be type-safe. The possibility of memory leaking on unique pointers seems to be a temporary trade-off of Cyclone. The focus operator in Vault is currently unsafe in multi-threaded environments.

6 Similarities and Dualities

6.1 Capability stores and dynamic regions

The C-style explicit memory deallocation causes dangling pointers. The idea of SafeC is to use a run-time database to track memory usage. Each newly allocated memory block is given a fresh capability (an unique identifier); each pointer is associated with such a capability; a set of "live" capabilities are maintained by the run-time system to represent the current memory layout. Accesses to pointers are guarded by checking the capability of the pointer in the current set of live capabilities.

In Cyclone, dynamic regions cause potential memory leaks: the state of each key (12 bytes) must be stored somewhere, holding the information of the dynamic region. Because keys are first-class values and they can be stored anywhere in the system, these states must be available even after the region has been freed. Hicks et al. [28] propose to solve this problem by using linear types. After studying SafeC, it appears that the dynamic mechanism in SafeC can be used to solve this problem easily. We hope this can be a contribution of this

paper. Dynamic regions can be treated as memory blocks in SafeC; the key can be simply represented as a capability. The capability store can be implemented as a hash table. If a capability is not in the hash table, it must have been freed, because each capability is *unique*.

Although SafeC has poor performance, the performance overhead of our approach in Cyclone is almost trivial for three reasons. First, it tracks regions instead of memory blocks. There are far fewer regions than memory blocks. Second, only one dynamic check is required for the whole lexical block using a dynamic region because all accesses inside the lexical scope are statically guaranteed to be safe by the type system. Third, although the pointer representation in SafeC is very verbose, the capability alone is enough to represent the region key.

6.2 Regions and linear types

Cyclone has a type system dedicated for region-based memory management. Vault has a linear type system expressive enough to support regions. In fact, these two type systems share a lot of similarities. Cyclone use lexically scoped region names to rule out the use of illegal region accesses, while Vault use key guards to rule out unsafe accesses. The lexical scope of a region name in Cyclone corresponds to the "live" area of some key in Vault, i.e. the program points where the required key is present in the held-key set. The lexical region handles in Cyclone and the region objects in Vault both have singleton types. The function types of Cyclone and Vault are both polymorphic with regard to region names and key names respectively. As a result, some Cyclone regions can be encoded by Vault. Vault provides more freedom on the use of regions as it does not require the FIFO ordering. Regions in Vault are lineally tracked while the dynamic regions in Cyclone has potential memory leaks via the status information.

That said, Vault does not surpass Cyclone in all aspects. The FIFO ordering of Cyclone regions allows the type system to perform finer analysis such as region subtyping. To simulate region subtyping in Vault, the type system must be able to reason about the relationship between key guards such as "for all program points in this scope, if guard A holds, then guard B holds". This may be worth thinking about as a future work.

Cyclone also does a better job modeling the dependencies among stack frames in C, whereas in Vault all pointers point to heap objects. The design of Cyclone is affected by many concerns on compatibility. While the Vault type system is elegant and expressive, it is built on the basis of a higher-level and more abstract language than C. Cyclone also provides unique pointers to manage resources linearly. It seems that Cyclone is trying to achieve some goals of Vault by using a different approach: rather than using a linear type system and encoding regions on top of linear types, Cyclone uses region types as its basis and attempts to express linearity using existing idioms for regions: a linear object is expressed in Cyclone as a fresh and unique region; the alias construct is cleanly implemented using lexical regions.

6.3 Adoption and focus, alias and swap

The relaxation mechanisms of linear types address interesting and realistic problems in Cyclone and Vault. Figure 14 shows the life cycles of a shared linear pointer in Cyclone. A shared linear pointer is protected by the type system and it cannot be directly accessed. The only way to use it is to swap it with a linear pointer. A linear pointer can be temporarily aliased in a lexical scope. As we can see in the figure, one problem is that there is no type-safe ways to create or destroy shared linear objects: the swap operation neither creates or consumes a linear pointer, so the type system can only assume that the shared linear object is available at the very beginning and remains available forever. At the current stage, it is the programmer's responsibility to ensure that such objects are properly initialized and recycled. Clearly, the linear aspects of Cyclone still require some improvement.

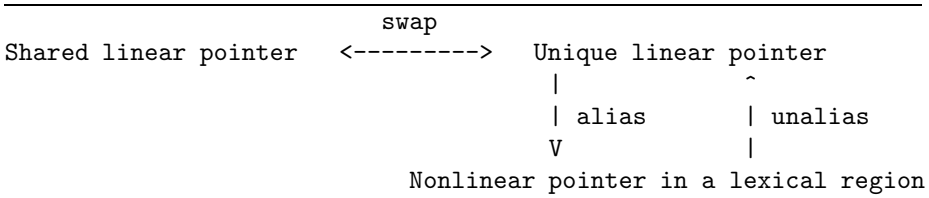

Figure 14: Life cycles of a shared linear pointer in Cyclone

Figure 15 shows the life cycles of a linear object in Vault. It is interesting that every object starts from a linear type. After adoption, the object becomes nonlinear, but guarded by the key of its adopter. In the run-time representation, it is linked to a hidden field of its adopter so that it can be implicitly deallocated together with its adopter. Guarded nonlinear objects can be temporarily made linear by using the focus operator.

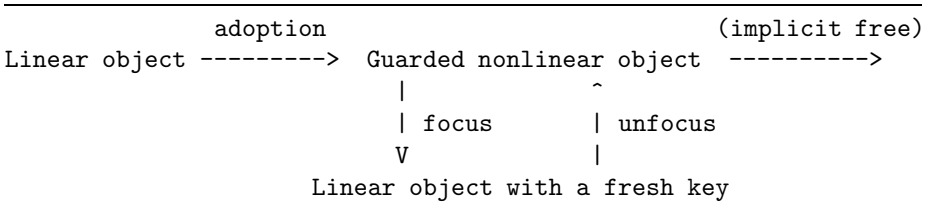

Figure 15: Life cycles of a linear object in Vault

These pictures again show the duality between Cyclone and Vault. Shared linear pointers in Cyclone are **linear** by default: they can be temporarily aliased in a lexical scope. Shared linear objects in Vault are **nonlinear** by default: they can be temporarily focused in a lexical scope. Cyclone uses run-time swaps; Vault uses hidden object fields in the run-time representation. Cyclone slightly compromises safety on creation and destruction of shared linear objects; Vault is unsound in a multi-threading setting because of its focus operation.

Figure 14 and 15 clearly shows the similarity between *focus* and *alias*: they are implemented in similar approaches. Cyclone implements *alias* by using a local region name that does not escape the scope; Vault implements *focus* by choosing a fresh key that does not escape the scope. The relationship between region names and keys has been discussed in Section 6.2.

7 Conclusion

This paper motivates the need for safe systems programming languages, defines the characteristics of such languages, identifies common kinds of safety violations, and compares some influential research projects in this field. The approach of each work is illustrated with code examples; strengths and limitations and discussed; comparisons among these works are made in various aspects including language features, costs, safety guarantees and level of soundness. Some interesting similarities and dualities among these systems are presented; some future research directions are tentatively identified.

References

[1] *Cyclone User's Manual*, 2004. `http://www.research.att.com/projects/cyclone`.

[2] Alexander Aiken, Manuel Fahndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1995.

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. of the '94 SIGPLAN Conference on Programming Language Design*, pages 290–301, 1994.

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

[5] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057:103–122, 2001.

[6] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

[7] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9), September 1988.

[8] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, 1993.

[9] 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.

[10] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[11] Satish Chandra and Thomas W. Reps. Physical type checking for c. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 66–75, 1999.

[12] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 262–275, San Antonio, Texas, January 1999.

[13] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. of the SIGPLAN Conference on Programming Language Design*, pages 59–69, Snowbird, UT, June 2001.

[14] Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. Technical Report UCSC-CRL-92-27, 1992.

[15] John R. Ellis and David L. Detlefs. Safe, efficient garbage collection for c++. Technical Report 102, DEC Systems Research Center, 1993.

[16] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, October 2000.

[17] 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 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, October 2001. Chateau Lake Louise, Banff, Alberta.

[18] David Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, 1996.

[19] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. of the SIGPLAN Conference on Programming Language Design*, pages 13–24, Berlin, Germany, June 2002.

[20] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 313–323. ACM Press, 1998.

[21] David Gay and Alex Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 70–80. ACM Press, 2001.

[22] Andrew Ginter. Design alternatives for a cooperative garbage collector for the c++ programming language. Technical Report 91/417/01, Department of Computer Science, University of Calgary, 1992.

[23] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[24] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 141–152. ACM Press, 2002.

[25] Reed Hastings and Bob Joyce. Purify: fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, 1992.

[26] Fritz Henglein. Global tagging optimization by type inference. In *Proc. 1992 ACM Conf. on LISP and Functional Programming (LFP), San Francisco, California*, pages 205–215. ACM Press, 1992.

[27] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 175–186. ACM Press, 2001.

[28] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe and flexible memory management in Cyclone. Technical Report CS-TR-4514, University of Maryland Department of Computer Science, July 2003.

[29] Suresh Jagannathan and Andrew K. Wright. Effective flow analysis for avoiding run-time checks. In *Proceedings of the Second International Symposium on Static Analysis*, pages 207–224. Springer-Verlag, 1995.

[30] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[31] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. *AADEBUG*, 1997.

[32] Stephen Kaufer, Russel Lopez, and Sesha Pratap. Saber-c: an interpreter-based programming environment for the c language. In *Proceedings of the Summer Usenix Conference*, pages 161–171, 1988.

[33] Samuel C. Kendall. Bcc: Runtime checking for C programs. *Proceedings of the Summer Usenix Conference*, 1983.

[34] Andreas Kind and Horst Friedrich. A practical approach to type inference for EuLisp. *Lisp and Symbolic Computation*, 6(1/2):159–176, 1993.

[35] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *2001 USENIX Security Symposium*, 2001.

[36] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. *Lecture Notes in Computer Science*, 2029:217–??, 2001.

[37] S. P. Midkiff, J. E. Moreira, and M. Snir. Optimizing array reference checking in java programs. *IBM Syst. J.*, 37(3):409–453, 1998.

[38] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 81–91. ACM Press, 2001.

[39] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, page 93. IEEE Computer Society, 1998.

[40] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 128–139, Portland, OR, January 2002.

[41] George C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 106–119, January 1997.

[42] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 229–243, Seattle, Washington, October 1996.

[43] Martin Odersky. Observers for linear types. In B. Krieg-Brückner, editor, *ESOP '92: 4th European Symposium on Programming, Rennes, France, Proceedings*, pages 390–407, New York, 1992. Springer-Verlag.

[44] H. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *Automated and Algorithmic Debugging*, pages 119–132, 1995.

[45] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[46] Mark Shields, Tim Sheard, and Simon L. Peyton Jones. Dynamic typing as staged type inference. In *Symposium on Principles of Programming Languages*, pages 289–302, 1998.

[47] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas W. Reps. Coping with type casts in c. In *ESEC / SIGSOFT FSE*, pages 180–198, 1999.

[48] Sjaak Smetsers, Erik Barendsen, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In *Dagstuhl Seminar on Graph Transformations in Computer Science*, pages 358–379, 1993.

[49] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proc. of the 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, 2000.

[50] Joseph L. Steffen. Adding Run-time Checking to the Portable C Compiler. *Software - Practice and Experience*, 1992.

[51] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with Regions in the ML Kit. Technical Report DIKU-TR-97/12, 1997.

[52] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. pages 188–201. ACM Press, January 1994.

[53] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[54] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Progarmming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[55] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symp. on Operating System Principles (SOSP)*, pages 203–216. ACM Press, December 1993.

[56] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, September 2000.

[57] David Walker and Kevin Watkins. On regions and linear types (extended abstract). In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 181–192. ACM Press, 2001.

[58] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. *ACM SIGPLAN Notices*, 36(3):166–178, 2001.

[59] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.

[60] Hongwei Xi. Imperative Programming with Dependent Types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santa Barbara, June 2000.

[61] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[62] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.