Cuckoo: a Language for Implementing Memory- and Thread-safe System Services

Richard West and Gary T. Wong
Computer Science Department
Boston University
Boston, MA 02215
{richwest,gtw}@cs.bu.edu

Abstract

This paper is centered around the design of a thread- and memory-safe language, primarily for the compilation of application-specific services for extensible operating systems. We describe various issues that have influenced the design of our language, called Cuckoo, that guarantees safety of programs with potentially asynchronous flows of control. Comparisons are drawn between Cuckoo and related software safety techniques, including Cyclone and software-based fault isolation (SFI), and performance results suggest our prototype compiler is capable of generating safe code that executes with low runtime overheads, even without potential code optimizations. Compared to Cyclone, Cuckoo is able to safely guard accesses to memory when programs are multithreaded. Similarly, Cuckoo is capable of enforcing memory safety in situations that are potentially troublesome for techniques such as SFI.

1 Introduction

In recent times, there has been a trend to employ commercial-off-the-shelf (COTS) systems for diverse applications having specific needs (e.g., in terms of real-time requirements). For this reason, extensible systems have been proposed [1, 13, 14], to allow the underlying system to be customized with services that are tailored to individual application demands. However, various techniques to enforce system safety have proven necessary, to prevent untrusted users from deploying potentially dangerous extension code in traditionally trusted protection domains such as the kernel. Such techniques include *sandboxing* [15, 12, 13], type-safe languages [5, 2, 10], proof-carrying codes [8], and hardware-support [3, 7]. While hardware approaches are generally less expensive than software safety techniques, they require features such as paging and segmentation that are not common to all architectures, or may be too coarse-grained for memory protection of small objects. In contrast, software techniques have largely focused on memory protection without considering other needs of service extensions (e.g., safe management of asynchronous threads of control, fine-grained memory allocation, and predictable real-time performance).

This paper, therefore, focuses on a novel language that provides memory safety, while ensuring controlled access to code and data in the presence of asynchronous threads of control. As with languages such as Cyclone [5], our language called "Cuckoo" is both syntactically and semantically similar to C. This has the advantage that legacy code can easily be translated, and new programs may be written in a manner familiar to many existing application and system developers.

The significant contributions of this paper include a description of the key design issues of Cuckoo with attention given to issues not easily supported by other software safety techniques. Specifically, we focus on the ability to: (1) ensure safe access to memory even when code may be executed by multiple threads, (2) provide fine-grained control over memory usage (not easily achieved by languages such as Java), and (3) achieve run-time performance close to that of untrusted C binaries.

The rest of the paper is organized as follows: Section 2 focuses on the design issues of Cuckoo with respect to memory safety, while thread safety issues are discussed in Section 3. Section 4 describes the performance of our prototype Cuckoo compiler for a series of applications, some of which employ multiple threads. For applications that are single-threaded, we compare the performance of Cuckoo to other approaches including Cyclone, software-based fault isolation [12] and C. Related work is described in Section 5 while conclusions and future work are covered in Section 6.

2 Memory safety

We describe a program as *memory safe* if it fulfills the following two conditions:

- It cannot write to nor read from any memory location which is not reserved for its use by a trusted runtime system (i.e., library or OS);
- It cannot jump to any location which does not contain trusted code (where trusted code is either code which is generated by a trusted compiler, or accessed via a designated entry point to the trusted runtime). Note that instruction boundaries must be respected when jumping to a location that *does* contain trusted code.

These conditions are very similar to traditional definitions of memory safety (such as those assumed by [4]), except for their dependence on a trusted runtime system. We claim that a correct compiler for the language we describe, in combination with such a trusted runtime, will produce only memory safe programs. In the remainder of this section we discuss various challenges that have affected the design of our memory safe language, called Cuckoo.

Dynamic memory allocation. For dynamic memory allocation, we do not allow `malloc`-style operations. Instead, we introduce the C++-style `new` operator to allocate memory large enough for an object of a specific type. The return value will be a pointer to an object of the specified type. The `new` operator declares a particular space on the heap to be available for reallocation to objects of either: (1) the same type, or (2) compatible types with the same or smaller storage requirements. Observe that (2) implies a matching `delete` operator may allow heap memory to be reassigned from one harmless type to another, thereby avoiding unnecessary growth in heap usage. Further details about harmless types are described in the Appendix. At this point the reader should simply note these semantics do not guard against incorrect program behavior but will ensure memory and type safety.

Stack safety. Stack safety concerns the potential problem arising from dereferencing pointers to automatic variables after returning from the function in which they were allocated. Additionally, overflow errors arise when a stack extends into a region beyond some defined limit. Normally, stack overruns are detected by the underlying hardware (e.g., using page-based virtual memory). However, a key design issue of Cuckoo is to detect stack overflow within memory regions where hardware protection is not available. One such example includes support for multiple thread stacks *within* a super-page (e.g., 4MB memory area) on a platform such as the Intel x86 [15, 11].

The stack overrun problem is, in general, undecidable. Hence, the Cuckoo approach involves runtime checks to throw an exception if stack overflow occurs. Unlike other memory-safe languages, such as Cyclone, which compile source code to another high-level language, Cuckoo generates native machine code. This enables Cuckoo to accurately track stack usage, based upon its knowledge of register versus stack allocation at code generation time. In Cyclone runtime checks for stack overruns are not possible, because secondary compilation (e.g., using `gcc`) on the compiler's output may use stack space in an unknown way.

In some cases, Cuckoo allows compile-time verification of stack usage when the maximum stack depth can be determined. In general, we have the potential to push up the call stack run-time checks to determine stack overruns, to points where all ensuing control-flow's stack usage can be verified, thereby reducing the number of runtime checks necessary. Note, however, that conditional branches that may affect stack usage ideally require runtime stack checks to avoid prematurely throwing stack overflow exceptions when in fact there is enough space available.

```
extern int a(...) { // suppose stack
                    // usage is small
                    // in this block

 char a_local;

 if (...)
   b();
}
static void b (...) { // again, have
                      // minimal stack usage
 if (...)
   c();
}
static int c() {
 char c_local[65536]; // stack-allocate
                      // lots of memory
 ...
}
```

Figure 1. Example to illustrate where stack checks must be performed

Further details can be found in the example shown in Figure 1. In this figure there is a limit on how far up the call stack we can place runtime checks for stack overflows. For example, placing a runtime check before the conditional (if statement) in function `b()` may incorrectly presume the following code will use as much as 64KB of stack space (given function `c()` is called) but this may not be the case, depending on the value of the conditional expression. Also, because function `a()` has an `extern` storage class specifier it must perform its own stack checking as it is generally unknown which control path has invoked function `a()`. In contrast, because functions `b()` and `c()` are `static`, it is possible to track where they are called from, thereby reducing runtime stack overflow checks to locations in calling functions, rather than the static functions themselves. However, to deal with general and mutual recursion, we must

perform a runtime stack check every iteration, as it is usually undecidable how many times the function will be invoked.

Pointers and array bounds checks. Cuckoo performs array bounds checks at compile-time for cases where the array size and index are both compile-time constants. All other cases involve runtime bounds checks. Our prototype implementation stores the array size in a memory location immediately preceding the first element of each dimension of the array stored in row major format. This ensures that the size of an array is always known at runtime. By comparison, in C, it may be the case that an array is used when its size is unknown. This has impact on the way Cuckoo treats arrays. Specifically, the name of an array in Cuckoo is a pointer to an array of some known number of objects of a given type, whereas in C an array name is a pointer to the first object in the array. This means that, whereas in C a variable of type "array of T" can be used interchangeably with a type "pointer to T", in Cuckoo the same variable can be used interchangeably with a "pointer to an array of T". This is discussed further in the Appendix with respect to "Pointer Generation". Figure 2 shows the similarities and differences between C and Cuckoo for array types.

In Cuckoo, any pointer object, P, is either NULL or points to storage which can be safely treated as having a type of $*P$, while P exists. In order to make automatic pointer validity tracking easier, Cuckoo differs from C with respect to the lifetime of local variables. In Cuckoo, the lifetime of locals is extended from the scope of a block to the scope of a function. With respect to stack safety discussed earlier, this approach also has the advantage that stack checks are simplified; only one check at the beginning of a function is necessary, to establish space for all automatic variables encountered in that function.

Dangling pointers. Dangling pointers are problematic in two situations: (1) dereferencing pointers to deallocated heap memory, and (2) dereferencing pointers into stack space on return from a function. In the latter case, the Cuckoo type system prevents assignment of an automatic object address, α, to a pointer whose lifetime is longer than α (see Appendix, Section A7.17).

Given the above similarities and differences between C and Cuckoo, Figure 3 compares the two languages in terms of legal casts. Further information can be found in the Appendix under "Pointers and Integers".

Exception handling. Currently all runtime exceptions trap to a default handler. A simple extension to this is to allow a generic signal (e.g., `SIGUSR1`) to be raised, thereby allowing application-code to customize the exception handling for specific cases. In future work, we plan to extend Cuckoo with exception-handling capabilities similar to those in C++ and Java.

Type homogeneity of dynamically allocated memory objects. As with the LLVM approach [4] to deal with pool-based dynamic memory allocation, Cuckoo allows heap memory to be reallocated (after a `delete`) to objects of compatible type. By "compatible" we mean that any aliases created by such an allocation cannot violate type-safety (thereby leading to potential memory protection problems). For example, Figure 4 shows how a freed heap memory area is reallocated to an object of a different type, thereby allowing an arbitrary memory location to be dereferenced. In Cuckoo, the statement `q=new(char *)` is guaranteed to return an address on the heap that is not incompatible with a `char *`, so it is impossible for the return value to be the same as previously assigned to `p`.

3 Thread safety

Many system services rely on execution models which are considerably more complicated than the traditional model of a single synchronous thread of control. The execution of concurrent threads and the processing of asynchronous signals and kernel upcalls mean that correctness guarantees are significantly more difficult to achieve. We do not (and cannot) prove the correctness of user code with respect to concurrency, but it is vital to ensure that the memory safety checks described in the previous section still hold.

We cope with null pointer checks by evaluating all pointer values in registers, so that the value checked is identical to the value dereferenced. Even if the code does not execute atomically, the register is guaranteed to be restored, so there cannot be a null pointer dereference. This does mean that we might dereference a pointer whose value is stale, as opposed to a dangling pointer. Observe that Cuckoo imposes no restrictions on pointer aliases, and non-atomic manipulation of pointers may lead to violations in program correctness but, nonetheless, will not violate memory safety as defined in Section 2.

The key issue here is that languages such as Cyclone support the use of fat pointers (i.e., pointers that maintain size information about the object they reference), but concurrent updates to such pointers can lead to race conditions and, hence, out-of-bounds memory accesses. In Cyclone, updates to pointer addresses are not made atomically with respect to changes (and checks) on the corresponding size field. For example, a fat pointer p shared by two threads may initially refer to an array $a[20]$ and later be updated to refer to a different array, $b[10]$. While the pointer may refer to the address of b its size field may be inconsistent, having the old value of 20, thereby enabling a potential array overflow on b. This situation is not possible with Cuckoo. Moreover, array bounds checks (as stated earlier) are trivial given that size information is embedded in the array objects

```
char a[5];

char c1 = *a;      // valid in C but not Cuckoo
char c2 = a[0];    // valid in Cuckoo, s.t. c2 is the same as c1 in C
char c3 = (*a)[0]; // also valid in Cuckoo: c3 is the same as c2
```

Figure 2. Array types in C versus Cuckoo

```
struct foo {
  int a[5];
  char *s;
}

struct foo *p;
int x = *((int *)p);       // legal in C but not in Cuckoo...
int y = *((int (*)[5])p);  // this is also illegal in Cuckoo,
                           // since we cannot assign an array to an int, but...
int z = ((int (*)[5])p)[0];// is legal in Cuckoo, as this assigns z
                           // the first element of an array
```

Figure 3. Example casts in C and Cuckoo

```
int *p;
char **q;

p = new(int);    // heap-allocate an integer
...
delete(p);       // release memory referenced by p
q = new(char *); // reuse heap memory freed at address p with
                 // incompatible typed object;
*p = 123;        // after freeing p, continue to assign values to its
                 // heap memory area

**q = 45;  // this results in writing the value 45 in address 123,
           // potentially violating memory safety
```

Figure 4. Potentially unsafe reallocation of freed heap memory

themselves, rather than carried along with pointers that reference them.

4 Experimental Results

The implementation details of Cuckoo are outside the scope of this paper. Currently, we have a version of the compiler that leverages the `gcc` preprocessor, linker and assembler. After preprocessing a source file, the cuckoo compiler performs compile-time checks on types, lifetimes of objects referenced by pointers, and certain array bound and NULL pointer checks, where the values are compile-time constants. Run-time checks are inserted for array and pointer accesses, where array indices cannot be determined within bounds at compile time, and pointers cannot be guaranteed valid. By valid we mean that a pointer is non-NULL and references an object of compatible type (considering type information includes the lifetime of objects). Observe that determining whether or not a pointer references a compatible type does not require run-time checks.

To date, we have a Cuckoo compiler that generates target assembly code for the 80386 architecture. While we currently use the GNU linker, it is quite possible for Cuckoo source files to be compiled and linked with untrusted objects, whose symbols may be referenced from Cuckoo code. While typing rules must be adhered in Cuckoo code that references external symbols in untrusted code, there is no guarantee of safety within the body of an external object. Future work includes the development of a trusted linker, which checks type-safety across separately compiled objects and detects attempts to link with untrusted code. Right now, Cyclone suffers from these same limitations.

Table 1 shows the performance of our prototype (compared to `gcc` 3.2.2 and Cyclone 0.7 running on a 2.8 GHz Pentium 4 CPU). The first benchmark, SUBSET-SUM, is a naive parallel solution to the well-known (and NP-

Compiler	Time (user)	Time (system)	Size (code)	Size (data)	Size (BSS)
SUBSET-SUM					
Cuckoo	30.96	n/a	2377	288	152
`gcc -O2`	17.86	n/a	1833	280	192
`gcc`	24.75	n/a	1945	280	192
PRODUCER-CONSUMER					
Cuckoo	2.50	5.13	2527	308	428
`gcc -O2`	2.46	5.10	2001	300	480
`gcc`	2.50	5.14	2093	300	480
FIND-PRIMES					
Cuckoo	10.17	n/a	1301	260	10016
Optimized Cuckoo	6.78	n/a	1285	260	10016
`gcc`	9.56	n/a	874	252	10032
`gcc -O2`	3.57	n/a	814	252	10032
Cyclone	12.43	n/a	91721	3340	59996
Cyclone `-O2`	5.51	n/a	91669	3340	59996
Cyclone `-noregions -nogc`	12.43	n/a	51619	2020	11140
Cyclone `-noregions -nogc -O2`	5.50	n/a	51567	2020	11140
SFI	10.79	n/a	970	252	10032
SFI `-O2`	4.31	n/a	858	252	10032
SFI protection	11.10	n/a	1058	252	10032
SFI protection `-O2`	4.32	n/a	870	252	10032

Table 1. Comparison of execution time and storage requirements

complete) decision problem of whether a given set of integers contains a subset which sums to zero. The results listed indicate the times (in seconds) and program size (in bytes) required to compute the SUBSET-SUM decision for a set of 27 random integers (the set was kept identical for each run and the numbers were in the range $[-10^6,+10^6]$), using 4 threads. The PRODUCER-CONSUMER test consists of one producer thread and one consumer thread, sharing a single memory buffer which is filled by the producer and emptied by the consumer.

The FIND-PRIMES benchmark uses a single-threaded implementation of the Sieve of Eratosthenes algorithm to compute a list of prime numbers (in our case, 10^5 iterations of the algorithm finding all primes below 10^4). Note that an "Optimized Cuckoo" entry has been added to this last result, which simulates the performance of a slightly more sophisticated compiler which honors the `register` keyword by keeping certain variables in CPU registers instead of main memory. Since our prototype compiler does not yet implement this optimization, the "Optimized Cuckoo" results were obtained by modifying Cuckoo assembly language output by hand to keep the values of index variables in the innermost loop in spare registers. The significant improvement in runtime from this small change seems to indicate that there is substantial room for improvement if an optimizing compiler were written for our language. Since the FIND-PRIMES benchmark is single-threaded, it is also compatible with the Cyclone 0.7 compiler. We have listed the results of four tests with Cyclone, with various compiler options: `-O2` invokes the optimizer in the `gcc` back-end, and `-noregions -nogc` disable Cyclone's region analysis and garbage collector (which are not required in this benchmark, as no runtime dynamic memory allocation is performed).

Lastly, the SFI results show the performance of software-based fault isolation [12], that inserts run-time checks on jumps and stores to memory addresses unknown at compile-time (e.g., those stored in registers). Since no SFI implementation exists for the x86 architecture, these benchmarks were compiled with `gcc` 3.2.2 and the assembly output of the compiler was modified by hand to insert the SFI-style address bound restrictions. The compiler's register allocations were modified by hand where necessary to preserve a sufficient number of reserved registers, and then extra instructions were added to memory accesses according to the SFI "address sandboxing" procedure. The `-O2` rows are the results when the original compiler was `gcc -O2`, and the other two rows were compiled with plain `gcc`. The "protection" rows implement full protection (by modifying both memory read and write operations), while the other two rows implement fault isolation only (where writes are sandboxed but no attempt is made to prevent illegal reads).

The results using SFI are slightly better than with Cuckoo. We conjecture the performance with SFI is partly due memory access time being the bottleneck, as opposed to CPU speed. The deep pipelining allows SFI memory checks to be performed (to some degree) in parallel with memory loads and stores. However, it should be noted that the hand-modifications used in the SFI tests do not deal with all safety issues, such as potential jumps to misaligned addresses in code segments, or preventing constant data within code segments from being executed. In contrast, Cuckoo prevents

pointer arithmetic and aliasing of pointers to incompatible types (e.g., aliasing to different members of unions). This eliminates the potential unsolved problems with SFI. Also observe that while SFI performs well for this example, it is quite possible that for many register-intensive applications, an architecture which is register-limited (e.g., the x86) may prove problematic. We plan to study more application examples in the future.

For the two CPU-intensive benchmarks (SUBSET-SUM and FIND-PRIMES), it appears that Cuckoo is on the order of 15% more expensive (in time and space) than unoptimized `gcc`. Some of this cost can be attributed to array bound and null pointer checks, which must be performed at runtime, and therefore contribute to both the execution time and code size, and part of the expense can be blamed on poor compiler implementation, since so far very little effort has been spent attempting to make our prototype compiler generate efficient code. However, note that in the second (PRODUCER-CONSUMER) benchmark, which is data- rather than CPU-intensive, the execution time for the Cuckoo code is comparable to that produced by `gcc`. Therefore, while we admit that for CPU-intensive or general purpose code we cannot realistically expect a Cuckoo compiler to match the efficiency of C, we hope that for the type of code in our target domain (low-level, data-intensive system services), the overheads we require will be minimal, and, depending on the environment, worth sacrificing for the safety guarantees we can provide.

To finish this section, we compared Cuckoo versus `gcc` for a parallel implementation of the subset-sum problem on a 4x2.2GHz AMD Opteron machine. Table 2 shows the real-time results. As expected, `gcc -O2` is the fastest but Cuckoo is slightly slower than unoptimized `gcc`. The difference is performance is arguably outweighed by the benefits of extra safety checks being performed by the compiler and, as stated earlier, there is room for improvement in the Cuckoo compiler's generated code. Notwithstanding, we believe this is evidence that a type-safe language can efficiently generate safe code with concurrent execution requirements.

Compiler	Parallel time (real)
Cuckoo	9.45
`gcc -O2`	4.59
`gcc`	7.40

Table 2. Execution times for parallel subset-sum problem

5 Related work

Table 3 compares Cuckoo to several notable software safety techniques. While other approaches include CCured [9], and Modula-3 [10], we feel characteristics of these languages are largely captured by those illustrated in the table. With Cuckoo: (1) there is no need for garbage collection, (2) correctness is not enforced (only safety), (3) runtime checks are allowed, (4) strict compatibility with C is sacrificed for ease of enforcing safety checks, and (5) thread safety is integral to the design of the language. Of the other related works, no other language or compiler provides this set of characteristics, which we feel is important for our target domain, namely, extensible system services. For example, Cyclone does not ensure multithreaded memory safety, and although Java provides this, it requires restricted memory management and corresponding garbage collection.

Of the techniques listed in Table 3, SFI appears to have the most similarities. However, SFI provides only partial memory safety compared to Cuckoo. To illustrate this, Figure 5 shows a situation that SFI cannot easily detect. In this example, there is a jump to an address that is 10 bytes from the beginning of function `bad` and this may not be the start of the instruction. Observe that this problem only exists on CISC architectures, that allow for variable-length instructions. SFI was originally designed for RISC architectures that have fixed-length instructions aligned on word boundaries.

```
static void bad( void ) {
    volatile int x = 0x0BADC0DE;
}

extern int main( void ) {
    union foo {
        char *data;
        void ( *code )( void );
    } bar;

    bar.code = bad;
    bar.data += 10; // whatever the offset is
                    // to 0x0BADC0DE
    bar.code();
    return 0;
}
```

Figure 5. Example jump to a potentially unaligned address

Finally, while there has been work on purely static analysis of program safety [4], such approaches are made more complicated when there are asynchronous execution paths, and have thus far applied control-flow analysis techniques to programs with single threads of execution.

System	C	Cyclone	Java	SFI	Cuckoo
Efficient memory use	✓	✓		✓	✓
Memory safe		✓	✓	partially	✓
Stack overflow checking			✓	✓	✓
Multithreaded memory safety			✓	✓	✓
Can operate without garbage collection	✓	✓		✓	✓
Unrestricted allocation without garbage collection	✓			✓	✓

Table 3. Features offered by various languages and approaches

6 Conclusions and Future Work

This paper presents an overview of the Cuckoo compiler and its design considerations with respect to supporting thread and memory safe programs. The specific application domain of this compiler is in support of system service extensions, where asynchronous control-flow (e.g., using threads and signals) is commonplace. Our insights gained from research on extensible operating systems [14] using type-safe languages such as Cyclone led us to believe that multithreaded memory safety was a key issue in extension code. Results show that Cuckoo provides program safety for both single- and multi-threaded applications with relatively low runtime overheads compared to comparable code compiled using unsafe C compilers, such as `gcc`.

Results show that, while techniques such as SFI have the potential to perform with low runtime overheads, it is arguably difficult to check for references to misaligned addresses and jumps to instructions embedded in data constants within program text segments. Cuckoo has the ability to check for these issues, using its typing rules at compile time.

Future work involves studying the costs of dynamic memory allocation, and its safe usage in the presence of asynchronous control-flow. Specifically, dynamic memory allocation within a signal handler requires mutually exclusive access to heap data structures, that maintain pools of free and allocated memory. Using traditional locking mechanisms to allocate and update the state of heap memory can potentially lead to deadlocks between signal handlers and a program's main thread of control holding a lock at the time a signal handler is invoked.

A Language Definition

The Cuckoo language is exactly the same as C as defined in the 2nd edition of K&R [6] Appendix A, with some notable exceptions. Below, we outline the differences between Cuckoo and C, labelling each of the following subsections with the corresponding appendix labels in the K&R book [6]:

A4.1 Storage Class

Automatic objects are local to a function, not to a block; they retain their values across exit from and reentry to blocks, and are not discarded until exit from the function.

> The main reason for this change is to ensure that local storage is never deallocated within a function (which prevents dangling pointers when one local points to another). Secondary benefits are that harmful objects (those which require initialized values for safety) need only be initialized once per function instead of once per allocation, and that stack growth within a function is reduced or eliminated, which reduces stack space checking overhead.

A4.5 Harmless Types *(new section)*

A type is "harmless" if it is a numeric (integral or floating) or void type, or if it is a structure or union which contains only harmless members.

> Conversely, "harmful" types are arrays, pointers, and any structure or union which (recursively) contains an array or pointer. Intrinsically, these types are only harmful when referenced by pointers that are cast to other types that could violate memory safety. Specifically, casting a pointer to a harmful type into a pointer to a harmless type can lead to: (a) dereferences of values used as addresses to invalid memory locations, and (b) overwriting values stored at the base addresses of arrays which are used for maintaining array size information. For example, the following code shows the potentially harmful effects casting $\&p$ which is a pointer to a harmful type, by assigning an arbitrary memory location, x, `some_value`:

```
int *p = some_address; // p is a harmful type
int *q;        // q points to a harmless type
q = (int *) &p;     // aliases a harmful type
*q = x;             // p = x
*p = some_value;    // memory[x] = some_value
```

A5. Objects and Lvalues

An lvalue also has a storage class; the permissible storage classes for lvalues are the two storage classes defined for objects (automatic and static) and also the class "unspecified". The discussion of each lvalue operator specifies which storage classes are permissible and which storage class is yielded. Expressions which are not lvalues do not have storage classes defined (except that expressions of array type do have storage classes, even though array types are not lvalues).

> Storage classes for lvalues are important because they allow us to detect potential references to deallocated automatic objects at compile time; see Section A7.17.

A6.6 Pointers and Integers

Addition and subtraction expressions involving pointer operands are prohibited.

> In K&R C, addition and subtraction expressions are not defined on pointers to objects which are not array elements; however, statically determining that a given pointer points to an array object is undecidable. We therefore forbid pointer arithmetic entirely. Similar results can be achieved in cases where the array object itself is known, by taking the address of an indexed array element.

Casts to or from pointers to functions are prohibited.

> This is required to ensure that the types of arguments match the types of function parameters when calling through function pointers.

Casts of other pointers are permitted only when casting: (1) to a numeric or void type, (2) to a pointer to void, (3) from a pointer p_i to a pointer p_j, where p_i and p_j point to harmless types h_i and h_j, respectively, and the size of h_j is no larger than the size of h_i, or (4) from a "pointer to `struct` S" to a "pointer to T", where T is the type of the first member of `struct` S.

> The result of case (1) is implementation-dependent (as in K&R C) but harmless. Case (2) is safe, because pointers to void may never be dereferenced, nor cast to any type which can be dereferenced. Case (3) can violate type safety only for harmless types, which are known not to violate memory safety. Case (4) is safe, being identical to K&R C as described in Appendix 8.3 (although it effectively forces the compiler to place the first member of a `struct` at the address of the `struct`).

Casts from integral types to pointers are prohibited, except that the constant zero may be cast to any pointer type to obtain a null pointer.

> When combined with the above restrictions, this limits the values that a pointer may hold to: (1) the null pointer, (2) the address of a compatible object obtained with the & operator, or (3) a value copied from a compatible pointer. All three cases are safe.

All pointers belong to one of two types: static pointers or automatic pointers. (The name refers to properties of objects the pointer may point to; it is unrelated to the storage class of the pointer itself.)

> This requirement allows us to determine the storage class of the lvalue obtained when the pointer is dereferenced.

Casts from automatic pointers to static pointers are prohibited.

> We must enforce the invariant that static pointers point only to objects with static storage class (see Section A7.17).

A6.8 Pointers to Void

Pointers to void may not be assigned to any other pointer nor cast to any other pointer type, although they are permitted in pointer equality comparisons.

A7.1 Pointer Generation

The value of an array expression is a pointer to the array, not a pointer to the first element of the array. If the storage class of "array of T" is static, then the type of the array expression is "static pointer to array of T", otherwise it is "automatic pointer to array of T".

A7.2 Primary Expressions

An identifier is an lvalue if it refers to an object and its type is arithmetic, structure, union or pointer; the lvalue storage class is the same as the object storage class.

The storage class of string literals is static.

The presence of parentheses does not affect the storage class of an expression.

A7.3.1 Array References

The first expression in an array reference must be of type pointer to array of type T, and the second must be integral.

> Array subscripting is no longer a commutative operation. Specifically, $E1[E2]$ is not equivalent to $E2[E1]$ for expressions $E1$ and $E2$.

The type of the result is T, and is an lvalue if type T is an arithmetic type, or a structure, union or pointer. The storage class of an array reference of type "static pointer to array" is static; the storage class of all other array references is unspecified (i.e., they could be static or automatic).

A7.3.2 Function Calls

Calls to undeclared functions are prohibited.

> This is because there is no way to check for type compatibility between arguments and function parameters for functions without prototypes.

A7.3.3 Structure References

The storage class of all structure and union member references is the same as the storage class of the structure or union expression.

A7.4.2 Address Operator

When applied to an object with static storage class, the '&' operator yields a static pointer; otherwise it yields an automatic pointer.

A7.4.3 Indirection Operator

When applied to an expression of type "static pointer to T", the storage class of the '*' operator is static; otherwise the storage class is unspecified.

A7.7 Additive Operators

Additive operations on pointers are prohibited.

> See Section A6.6.

A7.9 Relational Operators

Relational operations on pointers are prohibited.

A7.10 Equality Operators

Storage class is irrelevant when applying equality operators to pointers; that is, an automatic pointer to object O is considered equal to a static pointer to object O.

A7.16 Conditional Operator

If one of the second and third operands of the conditional operator is an automatic pointer and the other is a static pointer, then the result is an automatic pointer.

A7.17 Assignment Expressions

It is illegal to assign a automatic pointer to a static pointer. If the left hand side of an assignment expression has storage class unspecified, or static, and is not a harmless type, then the right hand side may not have automatic pointer type, nor be an aggregate type which (recursively) contains any automatic pointer type.

> These rules are fundamental in ruling out dangling pointers to automatic storage. Together, they enforce the invariant that no static object may hold the address of an automatic object.

A7.19 Constant Expressions

Initializers must evaluate either to a constant or to the address of a previously declared external or static object.

> K&R C allows initializers to evaluate to the address of an object plus or minus a constant, but in the general case such an initializer would violate type safety.

A8.3 Structure and Union Declarations

All members of unions must be of harmless type. Incomplete types are permissible, but any type which is used must be completely defined in the same translation unit.

A8.5 Declarators

The "*pointer*" syntax for declarators is extended to:

pointer:
* *type-qualifier-list*$_{opt}$
* *type-qualifier-list*$_{opt}$ *pointer*
@ *type-qualifier-list*$_{opt}$
@ *type-qualifier-list*$_{opt}$ *pointer*

> Note that this is the primary difference in *syntax* between K&R C and Cuckoo; the only other differences are the addition of two keywords, `new` and `delete`.

A8.6.1 Pointer Declarators

A pointer declarator with '*' declares an automatic pointer; a declarator with '@' declares a static pointer. It is prohibited to declare a static pointer to an automatic pointer, or any aggregate type which (recursively) contains any automatic pointer type.

> For example, the first of the following declarations is prohibited but the second is allowed:

```
char *@* p;
char *@@ q;
```

A8.6.3 Function Declarators

Old-style function declarations are prohibited. If no parameter type list appears in a function declaration, it is interpreted as an explicit declaration that a function accepts no parameters.

Variadic functions are prohibited.

> This is because we cannot check the types of arguments against function parameters in variadic functions.

Functions returning automatic pointers to objects are prohibited.

> This is to avoid returning addresses of objects whose lifetimes are confined to the scope of the returning function. However, returning static pointers to objects is allowed. This means the first of the following prototypes is illegal but the second is not:

```
int *myfunc( ...  );
int @myfunc( ...  );
```

The types of parameters that are arrays are altered to automatic pointers to arrays.

> For example, the following declarations are equivalent:
>
> ```
> int myfunc( int array[] );
> int myfunc( int (*array)[] );
> ```
>
> To explicitly declare a static pointer to an array, the following is allowed:
>
> ```
> int myfunc( int (@array)[] );
> ```

A8.7 Initialization

Automatic pointer objects which are not explicitly initialized are initialized to the same value that a static object of the same type would receive. Pointer members of aggregate objects are likewise initialized. This initialization occurs each time the enclosing function is called.

> Of course, an optimizing compiler would still be free to elide the implicit initialization if it could determine that the pointer is always initialized before being accessed.

A9.6 Jump Statements

It is illegal to return an expression of automatic pointer type.

> This restriction makes sure that the address of an automatic object is never accessible outside the function it is declared in (or that function's children): it cannot be returned (by this rule); it cannot be stored in a static object (see Section A7.17); and it cannot be stored in an automatic object in a calling function's context through a pointer parameter* (since the storage class of such an lvalue would be "unspecified"; see Section A7.4.3).
>
> *As an example of the latter case, the following code would generate an error, as variable `a` cannot store the address of variable `x`:
>
> ```
> int f2 (int **p) {
>  int x;
>  *p = &x;
>  ...
>  return 0;
> }
>
> int f1 (void) {
>  int *a;
>  f2(&a);
>  ...
> }
> ```

It is illegal to have an undefined return value in functions returning a harmful type.

A10.2 External Declarations

Incomplete arrays are forbidden in external declarations.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, and B. E. Chambers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[2] L. Carnahan and M. Ruark. Requirements for real-time extensions for the Java platform, September 1999. http://www.itl.nist.gov/div897/ctg/real-time/rtj-final-draft.pdf.

[3] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Symposium on Operating Systems Principles*, pages 140–153, 1999.

[4] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proc. Languages, Compilers and Tools for Embedded Systems 2003*, San Diego, CA, June 2003.

[5] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[6] B. W. Kernighan and D. M. Ritchie. *The C Programming Language Second Edition*. Prentice-Hall, Inc., 1988.

[7] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.

[8] G. C. Necula and P. Lee. Safe kernel extensions without runtime checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, 1996.

[9] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the Principles of Programming Languages*. ACM, 2002.

[10] G. Nelson. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, 1991.

[11] X. Qi, G. Parmer, and R. West. An efficient end-host architecture for cluster communication services. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster '04)*, September 2004.

[12] T. A. R. Wahbe, S. Lucco and S. Graham. Software-based fault isolation. In *Proceedings of the 14th SOSP*, Asheville, NC, USA, December 1993.

[13] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.

[14] R. West and J. Gloudon. 'QoS safe' kernel extensions for real-time resource management. In *the 14th EuroMicro International Conference on Real-Time Systems*, June 2002.

[15] R. West and J. Gloudon. User-Level Sandboxing: a safe and efficient mechanism for extensibility. Technical Report 2003-14, Boston University, 2003. Now revised and submitted for publication.