Memory Safety Without Runtime Checks or Garbage Collection *

Dinakar Dhurjati Sumant Kowshik Vikram Adve Chris Lattner
University of Illinois at Urbana-Champaign
{dhurjati,kowshik,vadve,lattner}@cs.uiuc.edu

ABSTRACT

Traditional approaches to enforcing memory safety of programs rely heavily on runtime checks of memory accesses and on garbage collection, both of which are unattractive for embedded applications. The long-term goal of our work is to enable 100% static enforcement of memory safety for embedded programs through advanced compiler techniques and minimal semantic restrictions on programs. The key result of this paper is a compiler technique that ensures memory safety of dynamically allocated memory *without programmer annotations, runtime checks, or garbage collection*, and works for a large subclass of type-safe C programs. The technique is based on a fully automatic pool allocation (i.e., region-inference) algorithm for C programs we developed previously, and it ensures safety of dynamically allocated memory while retaining explicit deallocation of individual objects within regions (to avoid garbage collection). For a diverse set of embedded C programs (and using a previous technique to avoid null pointer checks), we show that we are able to statically ensure the safety of pointer and dynamic memory usage *in all these programs*. We also describe some improvements over our previous work in static checking of array accesses. Overall, we achieve 100% static enforcement of memory safety without new language syntax for a significant subclass of embedded C programs, and the subclass is much broader if array bounds checks are ignored.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-based Systems; D.3 [**Software**]: Programming Languages; D.4.6 [**Software**]: Operating Systems—*Security and Protection*

Keywords

Embedded systems, compilers, programming languages, static analysis, security, region management, automatic pool allocation.

1. INTRODUCTION

Current and future embedded systems demand increasing software flexibility, including the ability to upgrade or introduce software modules into existing applications both offline and during active operation. Such software upgrades are becoming increasingly common for small consumer devices, and are expected to be important even for more constrained systems such as embedded control systems [23, 24] and sensor networks [20]. One of the key requirements for enabling dynamic software upgrades is to ensure that new software modules or applications do not compromise the safe and correct functioning of an embedded device. One part of this problem is ensuring the *memory safety* of embedded software, i.e., to guarantee that an upgraded software module cannot corrupt the code or data of its host application. (The term "memory safety" is defined in Section 2.)

Unfortunately, it appears that current language or system approaches for ensuring memory safety require significant overheads in terms of runtime checks and garbage collection. Safe languages like Java [10], Modula-3, ML, Safe-C [1], Cyclone [13] and CCured [22] use a variety of *runtime checks before individual memory operations* such as bounds checks for array references, null pointer references, and type conversions, and they rely on *garbage collection* to ensure the safety of references to dynamically allocated memory. The overheads of runtime checking are quite significant: languages like SafeC, CCured, Cyclone and Vault have reported slowdowns ranging from 20% up to 3x for different applications [1, 22, 11, 7].

Many types of embedded software must operate under stringent energy, memory, and processing power limitations, and often under hard or soft real-time constraints as well. The runtime overheads of safety checks and both the overheads and the potential unpredictability of garbage collection are unattractive for such software [4].

The long-term goal of our work is to ensure memory safety for embedded software while eliminating or greatly minimizing the need for runtime checks and garbage collection. Because this goal is impossible with ordinary language features such as arbitrary dynamic memory allocation, aliases, and array references, our strategy is to impose (minimal) semantic restrictions on programs necessary to achieve this goal with existing compiler technology, and to reduce these restrictions by developing new compiler techniques.

In previous work, we addressed the limited class of real-time control applications (which typically use very simple data structures and memory management) by designing a restricted subset of the C language appropriate for such pro-

*This work has been sponsored by the NSF Embedded Systems program under award CCR-02-09202 and supported in part by an NSF CAREER award, EIA-0093426.

LCTES'03, June 11–13, 2003, San Diego, California, USA.

grams. This language, which we called Control-C, imposed onerous restrictions on dynamic memory allocation, pointer usage, and array usage [15]. We showed that with these restrictions, *existing compiler technology* can permit 100% static checking of memory safety for this language. Unlike real-time control codes, however, other kinds of embedded applications use dynamically allocated memory and arrays in much more complex ways, and the restrictions in Control-C would preclude writing such applications.

A major technical challenge for broader classes of embedded applications is allowing flexible dynamic memory allocation and deallocation without runtime checks or garbage collection. Proving statically that a general C program (for example) never dereferences a freed pointer (the "dangling pointer" problem) is undecidable.

Several recent languages including Real-time Java [4], Cyclone [13], Vault [7], and others [26, 9, 5] have introduced language mechanisms for region-based memory management. In this approach, the heap is partitioned into separate regions and deallocation is only permitted on an entire region at once. These mechanisms guarantee the safety of pointer-based accesses to region data without garbage collection, but have two key disadvantages:

(1) Converting programs to use region-based mechanisms demands significant manual effort, typically requiring region annotations on pointer variables, function interfaces, and allocation sites.

(2) These region management schemes disallow explicit deallocation of individual objects, so data structures that must shrink and grow frequently (and have objects with non nested life times) have to fall back on a separate garbage-collected heap [4, 11] to avoid potentially unbounded growth in unused memory.

Automatic region inference algorithms have been developed that solve the first issue completely or partially, but only for languages with no explicit deallocation such as ML [25] and Cyclone [11] (i.e., languages which would otherwise require garbage collection).

Recently, we developed a fully automatic region inference algorithm called Automatic Pool Allocation that works correctly for C programs with explicit `malloc` and `free` (including non-type-safe programs) [17]. The transformation solves both the problems above because it is fully automatic and retains explicit deallocation of individual objects within regions. Each pool holds objects of a single size, thus eliminating fragmentation within pools and enabling very fast allocation and deallocation. Unfortunately, allowing individual object deallocation means that the transformation does *not* ensure memory safety (it makes no attempt to eliminate or restrict dangling pointers).

The main result of the current work is to extend Automatic Pool Allocation to ensure memory safety without requiring runtime checks, while retaining the performance and memory benefits of the original algorithm (including fully automatic region management without language annotations, and explicit deallocation of objects within regions). The key to our approach is that we do *not* prevent uses of dangling pointers to freed memory (e.g., a read, write, or free on an already freed storage location); instead, we ensure statically that such operations cannot cause violations of type safety or memory safety.

The specific contributions of this paper are as follows:

(a) We show how to use the Automatic Pool Allocation transformation to ensure the safety of references to dynamically allocated memory. We use an interprocedural flow analysis of the compiler-generated pool operations to pinpoint data structures for which our approach could lead to increased memory consumption by the program. The analysis also identifies pools for which individual object deallocation can be completely eliminated, without increasing memory consumption.

(b) We extend our previous program restrictions and compiler analysis for ensuring array access safety in order to support some common features that were disallowed before, including string manipulation, standard I/O operations, and argument vectors.

(c) We evaluate how effective the new techniques in this paper (and other compiler safety checks retained from the previous work) are in permitting static checking of memory safety, using a diverse collection of embedded programs from two widely used benchmark suites, MiBench [12] and MediaBench [19].

Our experimental results show that we are able to statically ensure the safety of pointer and dynamic memory usage *in all these programs.* Our compiler analysis identifies specific data structures in two of these programs where our memory management strategy could lead to some (probably small) potential increase in memory consumption. Overall, the approach promises safe dynamic memory management without the overheads of runtime checks, and with negligible manual programmer effort. We believe that this is a major step towards achieving our long-term goal of 100% static enforcement of memory safety for embedded programs.

Our results also show that two other essential techniques developed in our previous work are adequate for the programs studied here: a novel memory initialization strategy to eliminate runtime checks for null pointer references under a specific system assumption, and a compiler analysis for lifetimes of stack-allocated data. The former is sufficient for all our programs (and necessary for most), while the latter works successfully for 16 out of 17 programs. Proving the safety of array references, however, remains a major challenge in achieving our long-term goal. With the extensions above, we are able to prove the safety of array references in 8 out of 17 of these programs. We draw some ideas for future language and compiler mechanisms that might succeed for the other programs.

The rest of this paper is organized as follows. The next section defines what we mean by memory safety and static checking, briefly describes our previous work on static safety checking for real-time control systems, and summarizes the assumptions we make about programs and systems in our current work. Section 3 describes the language restrictions and compiler techniques for ensuring safety of pointer references and heap management. Section 4 does the same for array references. Section 5 describes our experiments evaluating the effectiveness of our techniques in supporting different classes of embedded applications. Section 6 compares our work with previous work on providing program safety through static techniques. Section 7 concludes with a summary of our results and suggests directions for further research.

2. DEFINITIONS, PRIOR WORK, AND ASSUMPTIONS

For the purposes of this work, we define a software entity (a module, thread, or complete program) to be *memory-safe* if (a) it never references a memory location outside the address space allocated by or for that entity, and (b) it never executes instructions outside the code area created by the compiler and linker within that address space. In practice, to enable static enforcement of the above requirements, we must enforce stronger restrictions, e.g., strict type rules for all operations, limited type conversions, and in-bounds array accesses. The stronger restrictions also help to detect many kinds of errors at compile-time rather than at runtime.

By “static enforcement” or “static checking” of memory safety, we mean that the compiler must ensure memory safety without relying on garbage collection and without introducing *any* runtime checks before program operations (e.g., null pointer, array bounds, or type conversion checks). Some runtime support is still necessary, especially initialization of global or dynamically allocated storage, and some system assumptions for error recovery described later.

Many ordinary constructs in modern languages (particularly array accesses and complex pointer-based data structures) make it impossible to ensure memory safety via static checking alone. We therefore choose to impose some restrictions on programs to make static checking possible. To make it as simple as possible to modify existing embedded code to conform to our restrictions, we avoid adding *any* new language mechanisms or syntax. Instead, we impose usage (i.e., semantic) restrictions that can be defined within the framework of an existing language, and checked by a compiler.

Although our experiments focus on C programs in this paper, our semantic restrictions are defined in low-level language-independent terms, and our safety checking compiler is implemented entirely in a language-independent compiler infrastructure called LLVM (Low Level Virtual Machine) [16][1]. These features, together with the lack of any new source-level constructs, imply that our safety-checking strategy can be used for programs in any source-level language compiled to LLVM object code.

2.1 Control-C: Memory Safety for Real-Time Control Codes

The Control-C language defined in our previous work [15] imposed many semantic restrictions on C programs, and added one new language mechanism for manual region allocation (described below). The goal was to enable 100% static checking of memory safety for real-time control codes using *existing* compiler techniques. We briefly describe that language to provide a basis for understanding how new compiler techniques in the current work can eliminate the need for some of these restrictions.

There were 3 classes of restrictions in Control-C:

- *Type safety*: Input programs had to be strongly typed. These restrictions are retained in our current work.
- *Affine Expressions for Accessing Arrays*: Control-C imposed restrictions on array index expressions and loop bounds so that *the net effect* is to produce an affine relationship between the effective address expression and the size expression for each array dimension. It also disallowed most common library functions and string operations. The affine requirement is retained in this work but a number of trusted library functions and string operations are now permitted.
- *Single-Region Dynamic Memory Allocation*: Control-C imposed onerous restrictions on dynamic memory allocation and pointer usage, *all of which have been eliminated in the current work*:
  - Heap allocation was restricted to *a single region at a time*, and the entire region (i.e., *all heap objects*) had to be deallocated simultaneously.
  - Every time the region was freed, all scalar pointers variables (local and global) had to be reinitialized before their next use.
  - Structures or arrays containing pointers had to be allocated dynamically, either on the heap or on the stack using C’s `alloca` intrinsic.

We showed that these restrictions are adequate for many real-time control algorithms, which tend to use very simple data structures and memory management. They are clearly inadequate for broader classes of embedded programs. Eliminating these restrictions and retaining static checking of memory safety requires new compiler techniques, particularly for array bounds checking and for heap and pointer safety. The current work primarily focuses on the latter, while making some simple improvements to the former.

2.2 Assumptions of this Work

The system assumptions of the current work, plus the basic language restrictions for type safety and pointer safety are summarized below. The language restrictions for array safety are described in later sections.

First, we make some assumptions about the runtime environment. We assume that certain runtime errors are safe, i.e., the runtime system can recover from such errors by killing the applet, thread, or process executing the untrusted code. We assume a safe runtime error is generated if either the stack or the heap grows beyond the available address space. We assume the system has a *reserved address range* and any access to these addresses causes a safe runtime error, typically triggered by a page fault handler or by a reserved address range in hardware on systems without virtual memory management.[2] If this is not available, some null pointer checks must be inserted in the code, as described later.

We assume that certain standard library functions and system calls are trusted and can be safely invoked by the untrusted code (calls whose arguments must be checked are discussed in Section 4). We assume (and check) that the source code of all other functions is available to the compiler. We also require that the program be single-threaded.

We retain the basic type rules of Control-C, summarized informally below. We assume a low-level type system including a set of primitive integer and floating point types, arrays, pointers, user-defined records (structures), restricted union types, and functions.

[1] LLVM defines a simple, fully typed instruction set based on Static Single Assignment (SSA) form as the input code representation in order to enable compile-time, link-time and runtime optimization of programs. See `llvm.cs.uiuc.edu`.

[2] For example, in standard Linux implementations, the high end of the process address space is reserved for the kernel, typically 1 GB out of 4 GB.

(T1) All variables, assignments, and expressions must be strongly typed.

(T2) Casts *to* a pointer type from any other type are disallowed. (Certain pointer-to-pointer casts for compatible targets are considered safe, however.)

(T3) A union can only contain types that can be cast to each other, e.g., a union cannot include a pointer and a non-pointer type.

Enforcing the above rules is trivial in LLVM [16], where all operations are typed and only an explicit `cast` instruction can be used to perform any type conversion.

We also retain some rules required for ensuring pointer safety, which are discussed in Section 3:

(P1) Every local pointer variable must be initialized before being referenced, i.e., *before being used or having its address taken.*

(P2) Any individual data type (i.e., not an array) should be no larger than the size of the reserved address range.

(P3) The address of a stack location cannot be stored in a heap-allocated object or a global variable, and cannot be returned from a function.

3. SAFETY OF POINTER REFERENCES

In a language without garbage collection, and with the type restrictions **T1** − **T3** above, there are three key ways in which pointer usage can lead to unsafe memory behavior: (a) Uninitialized pointer variables (either scalars or elements of aggregate objects) could be used to access invalid memory addresses. (b) A pointer into the stack frame of a function that is live after the function returns could be used to access an object of a different type (i.e., to violate type safety). (c) A pointer to a freed memory object (a "dangling pointer") could be used to access an object of a different type allocated later.

These three problems must be detected and disallowed at compile-time or safely tolerated at runtime without introducing checks for individual memory references. We examine each of these conditions in turn in the following subsections.

3.1 Uninitialized pointers

Our compiler prevents the first error above (due to uninitialized pointer values) in the same way as in our previous work, through a combination of static analysis and minimal runtime support [15]. We describe this briefly here for completeness.

First, we use a standard global dataflow analysis to check rule **P1** above, which requires that all automatic scalar pointers must be initialized within their parent function explicitly before they are dereferenced or their address taken [15].

Detecting uninitialized values for global variables and for pointers within dynamically allocated data (e.g., structure fields or arrays), is difficult at compile-time. In order to avoid runtime null pointer checks, we initialize all uninitialized global scalar pointers and all pointer fields in dynamically allocated data structures at allocation time to point to the base of the reserved address range (analogous to the initialization of fields in Java). Pointer fields in stack-allocated variables of aggregate types are also initialized to the same value. Finally, the constant 0 used in any pointer-type expression is replaced with the same value. Rule **P2** above specifies that the size of any individual structure type[3] cannot exceed the size of the reserved address range. With this rule, the above initialization ensures that the effective address for the load of any scalar variable or structure field using an uninitialized pointer will fall within the reserved address range, thus triggering a safe runtime error. If a reserved address range is unavailable or the structure size restriction above is unacceptable, then runtime checks for null pointer references would be required.

3.2 Stack safety

Problem (b) above potentially arises when the address of a local variable (i.e., a pointer into the current stack frame) is made accessible after the function returns. To avoid this problem, many type-safe languages like Java disallow taking the address of local variables. We choose to be less restrictive: we only disallow placing the address of a stack location in any heap location or global variable, or returning it directly from a function (rule **P3** above).

Enforcing this rule requires sophisticated compiler technology, but no more than that required to perform Automatic Pool Allocation for enforcing heap safety. In particular, we use *Data Structure Analysis*, a flow-insensitive, context-sensitive (but very fast), interprocedural analysis that computes a *Data Structure Graph* for each procedure [18]. This is a directed graph of all the memory objects accessible within a procedure, along with their types, their storage class (stack, heap, global, formal parameter, return value, or local scalar temporaries), and the "points-to" links between them. The graph for each function includes reachable objects passed in from callers or returned from callees.

Rule **P3** can now be enforced using a simple traversal of the Data Structure Graph for each function, checking whether any stack-allocated object is reachable from the function's pointer arguments, return node or globals.

3.3 Heap Safety

The third error above, that of detecting unsafe accesses to freed memory, is a particularly challenging problem for a language with explicit memory deallocation. The example in Figure 1 illustrates the challenges. Function `f` calls `g`, which first creates a linked list of 10 nodes, initializes them, and then calls `h` to do some computation. `g` then frees all of the nodes except the head and then returns. `f` then uses a dangling pointer reachable from the head. In such code, it is extremely hard for any compiler to statically identify which references (if any) may be unsafe and which are not. Moreover, consider `h()`, which allocates one node and frees one node of the list 10^4 times. Eliminating explicit frees by using region allocation (such as in Control-C, Cyclone, or other languages with nested regions) would increase the instantaneous memory consumption of the program by 10^4 `* sizeof(struct s)` bytes because the region holding list items can be freed only after exiting the function `f`.

The key principle underlying our approach is the following: *(Type homogeneity principle) If a freed memory block*

[3] An array does not need this size restriction. An uninitialized pointer used as an array reference will be caught by the array bounds checker since the array will have no known size expression.

```
f() {                                 g(struct s *p) {
  ...                                   create_10_Node_List(p);
  g(p);                                 initialize(p);
  // p->next is dangling                h(p);
  p->next->val = ... ;                  free_all_but_head(p);
}                                     }

h(struct s *p) {
  for (j=0; j < 100000; j++) {
    tmp = (struct s*) malloc(sizeof(struct s));
    insert_tmp_to_list(p,tmp);
    q = remove_least_useful_member(p);
    free(q);
  }
}
```

Figure 1: Pointer safety and pool allocation example

holding a single object were to be reallocated to another object of the same type and alignment, then dereferencing dangling pointers to the previous freed object cannot cause a type violation. This principle implies that to guarantee memory safety, we do not need to prevent dangling pointers or their usages in the source – we just need to ensure that they cannot be dereferenced in a type unsafe manner. The principle allows correct programs (*i.e.* programs with no uses of dangling pointers), to work correctly without any runtime overhead. Programs with dangling pointer errors will execute safely but we cannot (and do not need to) prevent such errors for these programs.

Using the above principle directly, one simple but impractical solution is to separate the heap into disjoint pools for distinct data types and never allow memory used for one pool to be reused later for a different pool. This is impractical because it can lead to large increases in the instantaneous memory consumption. The worst-case increase for a program with N pools would be roughly a factor of $N - 1$, when a program first allocates data of type 1, frees all of it, then allocates data of type 2, frees all of it, and so on.

Our solution is essentially a more sophisticated application of this basic principle, using Automatic Pool Allocation to achieve type-homogeneous pools with much shorter lifetimes in order to avoid significant memory increases as far as possible.

3.3.1 Background: The Automatic Pool Allocation Transformation

The Automatic Pool Allocation transformation was developed as a general compiler technique for enabling *macroscopic* optimizations on logical data structures [17]. This transformation introduces pool-based memory management for a subset of the disjoint data structures in an ordinary imperative program that uses explicit allocation (e.g. `malloc`) and deallocation (e.g., `free`). It rewrites the allocation and deallocation operations to use separate pools of memory for each logical data structure instance (e.g., a particular linked list or a graph) that is not exposed to unknown external functions. A pool is created before the first allocation for its data structure instance and destroyed at a point where there are no accessible references to data in the pool.

We use a pool allocation library with five simple operations: (a) `poolinit(Pool** PP, TypeDesc* TD)` creates a new pool for objects of the specified type. (b) `pooldestroy(Pool* PP)` destroys a pool and releases its remaining memory back to the system heap. (c) `poolalloc (Pool* PP)` and `poolallocarray(Pool* PP, int N)` allocate a single object or an array of N objects in the pool. (d) `poolfree (Pool* PP, T* ptr)` deallocates an object within the pool by marking its memory as available for reallocation by `poolalloc` or `poolallocarray`. The pool library internally uses ordinary `malloc` and `free` to obtain memory from the system heap and return it when part of a pool becomes unused or the pool is destroyed.

The pool allocation transformation operates as follows:

1. *Identify data structure (DS) instances*: We traverse the Data Structure Graph of each function (described in Section 3.2) to identify maximal connected subgraphs containing only heap nodes. Each such subgraph represents a distinct heap-allocated data structure.

2. *Identify where to create/destroy pools*: For each procedure, the DSG can be used to identify those data structures that are not accessible after the procedure returns (i.e., do not "escape" from the procedure to its callers). For each such data structure, we insert calls to create and destroy pools of memory (one pool per data type used in the data structure) at the entry and exit of the procedure.[4] In our running example, the linked list does not escape from the procedure `f()` to its callers and so we create and destroy the pool for the list in procedure `f()`, as shown in Figure 2.

3. *Transform (de)allocation operations and function interfaces*: We transform all `malloc` and `free` calls in the original program to use the pool allocation versions, as illustrated in function `h()`. For any function containing such operations on a pool created outside the function, we add extra arguments to pass the appropriate pool pointers into the function (and do the same for possible callers of such functions, and their callers and so on).[5] This is illustrated by the functions `g()` and `h()` and their invocations in Figure 2.

The result of this transformation for type-safe programs is that all heap-allocated objects are assigned to type-homogeneous pools, disjoint data structure instances (as defined above) are assigned to distinct sets of pools, and individual items are allocated and freed from the individual pools at the same points that they were before. A pool is destroyed when there are no more live (i.e., reachable) references to the data in the pool.

Note that the transformation as described so far *does not ensure program safety.* Explicit deallocation via (`poolfree`) can return freed memory to its pool and then back to the system, which can then allocate it to a different pool. Dangling pointers to the freed memory could violate type safety.

3.3.2 Exploiting Pool Allocation for Heap Safety

The basic principle of type homogeneity mentioned earlier can be applied to ensure program safety after the pool

[4] Our pools do not require nested lifetimes. We could move `poolinit` later in the function and move the `pooldestroy` earlier or into a callee using additional flow analysis, but we do not do so currently.

[5] Data Structure Analysis also identifies the targets of function pointers and constructs a call graph, allowing us to handle programs with indirect calls and recursion.

```
f() {
  Pool *PP;
  poolinit(PP, struct s);
  ...
  g(p, PP);
  // p->next is dangling
  p->next->val = ... ;
  pooldestroy(PP);
}

g(struct s *p, Pool *PP) {
  create_10_Node_List(p, PP);
  initialize(p);
  h(p, PP);
  free_all_but_head(p, PP);
}

h(struct s *p, Pool *PP ) {
  for (j=0; j < 100000; j++) {
    tmp = poolalloc(PP);
    insert_tmp_to_list(p, tmp);
    q = remove_least_useful_member(p);
    poolfree(PP, q);
  }
}
```

Figure 2: Example after pool allocation transformation

allocation transformation. Since our pools are already type-homogeneous, we simply need to ensure that the memory within some pool P_1 is not used for any other dynamically allocated data (either another pool P_2 or heap allocations within trusted libraries) until P_1 is destroyed. This can be done easily by modifying the runtime library so that memory of a pool is not released to the system heap except by pooldestroy. This change can have the same disadvantage as the naïve type-based pools – the memory requirement of the program could increase significantly.

Note, however, that our pools are much more short-lived than in the naïve approach and are tied to dynamic data structure instances in the program, not static types. We expect, therefore, that during the lifetime of a pool, the most important reuse of memory (if any) is *within* the pool rather than between the pool and other pools. Only the latter causes any potential increase in memory consumption. Nevertheless, any such increases are likely to be of significant concern to programmers of embedded systems.

The goal of our further analysis is to distinguish the situations outlined above, and inform the programmer about data allocation points where potential memory increases can occur. We can classify each pool P into three categories:

Case 1 (No reuse): Between any poolfree for pool P and the pooldestroy for P, there are no calls to poolalloc from any pool including P itself. In this case, there is no reuse of P's memory until P is destroyed. Figure 3(a) illustrates this situation. Note that all poolfree calls to P can be *eliminated as a performance optimization.* This is essentially static garbage collection for the pool since its memory is reclaimed by the pooldestroy introduced by the compiler.

Case 2 (Self-reuse): Between any poolfree operation on pool P and the call to pooldestroy for P, the only poolalloc operations are to the same pool P. In this case, the only reuse of memory is within pool P, and the explicit deallocation via poolfree ensures that no increase in the program's memory consumption will occur. This is illustrated in Figure 3(b): after the first poolfree on p1 there are new allocations in pool p1 (via the function addItems), but not by any other pool.

Case 3 (Cross-reuse): Between the first poolfree operation on P and the pooldestroy for pool P. there are poolalloc operations for other pools. Pool $p1$ in Figure 3(c) falls in this category because there are allocations from pool p2 via the call to *addItems(p2,t).* Our transformation in this case may lead to increased memory consumption, and we require this to be approved by the programmer via a compiler option. In such situations, a programmer should be able to estimate the potential memory increase through manual analysis or profiling. In practice, we expect the amount of memory released by one pool and used by another, before the first pool is destroyed, will be relatively small.

Note that the pool in our running example of Figure 2 has only self-reuse, and we can guarantee memory safety without any increase in memory consumption. Our experiments in Section 5 have produced very few instances of case 3, and in only 2 out of the 17 embedded codes we examined.

3.3.3 Compiler Implementation

The compiler first applies the type checking, stack safety, and array safety analyses to the original program. It then applies Automatic Pool Allocation to transform the program as described earlier. We have modified our runtime pool allocation library so it does not release free memory in a pool back to the system heap until the pool is destroyed. The key goal of the new compiler analysis is to identify situations where this can lead to a potential increase in memory consumption by categorizing pools as described above.

Categorizing pools requires analyzing the potential order of execution of pool operations *across the entire program*, using an interprocedural control flow analysis. Automatic Pool Allocation records information about the pools used in each function and the locations of calls to poolalloc, poolfree and pooldestroy inserted for each pool. Pool pointers are passed between procedures but they are not otherwise copied and their address is never taken, so each pool pointer variable within a function identifies a unique pool. Our current pool allocation transformation places the calls to pooldestroy at the end of the function containing the call to poolinit for that pool.[6]

The algorithm for identifying and categorizing reuse within and across pools is shown in Figure 4. We say a function F (or a call site C) indirectly calls a pool operation (e.g., poolfree) if it calls some function that may directly or indirectly call that operation. The sets FreeSites(F,P) and AllocSites(F,P) respectively identify the call sites within function F that directly or indirectly invoke poolfree and poolalloc on pool P. The sets PoolsFreed(F) and PoolsAlloced(F) respectively are sets of incoming pools (i.e., formal pool pointer arguments to function F) for which F may directly or indirectly call poolfree or poolalloc.

Consider first a single-procedure program containing calls to poolfree, poolalloc and pooldestroy. The analysis then traverses paths from a poolfree for a pool to the unique pooldestroy of that pool, looking for all calls to poolalloc that appear on such a path. This is shown as routine AnalyzeFunction in Figure 4. (It is easy to handle all pools in a single linear-time traversal of the Control Flow Graph, but the version in the figure is much easier to understand.) Each pool is then categorized according to what instances of poolalloc, if any, are found on such paths.

Consider next an input program without recursion. The algorithm then makes a bottom-up traversal of the call

[6] The algorithms described in this section can be easily modified if poolinit and pooldestroy calls are placed differently by Pool Allocation.

```
p1 = poolinit(s);
t = makeTree(p1);
while(...) {
   processTree(p1,t);
   freeSomeItems(p1,t);


}
freeTree(p1,t);
poolDestroy(p1);
```

(a) **No reuse (case 1)**

```
p1 = poolinit(s);
t = makeTree(p1);
while(...) {
   processTree(p1,t);
   freeSomeItems(p1,t);
   addItems(p1,t);  // self-reuse

}
freeTree(p1,t);
poolDestroy(p1);
```

(b) **Self-reuse (case 2)**

```
p1 = poolinit(s);
t = makeTree(p1);
while(...) {
   processTree(p1,t);
   freeItems(p1,t);
   addItems(p1,t);  // self-reuse
   addItems(p2,t);  // cross-reuse
}
freeTree(p1,t);
poolDestroy(p1);
```

(c) **Self- and Cross-reuse (case 3)**

Figure 3: Example illustrating 3 types of reuse behavior for a pool p1.

graph, computing the four kinds of sets above for each function. The bottom-up traversal ensures that the sets PoolsFreed(C) and PoolsAlloced(C) will be computed for all possible callees C of a function F, before visiting F. To compute the sets for F, we visit each call site S in F and add this call to FreeSites(F,P) if it causes an invocation of poolfree(P), and to AllocSites(F,P) similarly. We also add each pool so encountered to PoolsFreed(F) or PoolsAlloced(F). We can now invoke AnalyzeFunction(F) directly to classify all pools in F. Note that AnalyzeFunction(F) makes no distinction between local and indirect calls to poolfree/poolalloc for pool P since both kinds of call sites are included in FreeSites(F,P) and AllocSites(F,P).

To handle recursive and non-recursive programs uniformly, we actually perform the bottom-up traversal on the Strongly Connected Components (SCC's) of the call graph. Within each SCC, we use a simple iterative algorithm in which the sets are propagated from a function to its call sites *within* the SCC until the sets FreeSites(F,P) and AllocSites(F,P) stabilize for all functions F in the SCC and every pool P. Once they have stabilized, the sets can be propagated from each function in the SCC to every call site of that function outside the SCC. AnalyzeFunction is then applied to each function F in the current SCC as explained earlier.

4. ARRAY RESTRICTIONS

In general, array bounds checking in general programs is undecidable. In our previous work [15], we designed language restrictions on array usage (rules (A1–A3) in Fig. 5) that enable complete symbolic checking of array accesses. Restriction **A3** says that every index expression in an array reference must have a provably affine relationship to the allocated array size for that dimension. We also described an interprocedural constraint propagation algorithm that propagates affine constraints on integer variables from callers to callees (for incoming integer arguments and global scalars) and from callees to callers (for integer return values and global scalars), as described in [15]. We can then perform a symbolic bounds check for each index expression using integer programming (our compiler uses the Omega Library from Maryland [14]).

For array safety, our primary goal in this work has been to evaluate the adequacy of these rules for a broad range of embedded programs, and to relax the rules in limited ways that can still be checked with existing compiler and integer programming technology. We have found (not surprisingly) that embedded codes typically use arrays in much more complex ways than the control codes studied in our previous work, as our experimental results in Section 5 show.

One practical issue for embedded programs is that they make significant use of I/O operations, the string library, and command line arguments. We added rule **(A4)** in Fig. 5 to allow certain trusted string and I/O library routines. The rule also specifies that the arguments to trusted library routines must satisfy some safety preconditions, to prevent buffer overruns within the library routines. Some library routines also provide constraints relating the output of the routine to its inputs which must be used by the compiler to check buffer or string safety. For example, the expression n = read(fd, buf, count) where buf is a character array has the safety precondition, (buf.size >= count) and a constraint on the return value, (n <= count) since read can only read up to count bytes. Some trusted library calls and the corresponding constraints are listed in Figure 6.

Library Call	**Return Value Constraints**	**Safety Preconditions**
n = read(fd, buf, count)	n <= count	buf.size >= count
n = puts(s)	-	-
p = memcpy(p1, p2, n)	p.size = p1.size	p1.size >= p2.size
fp = fopen(p,m)	-	-
n = getc(s)	-	-
n = strlen(s)	n < s.size	-
p = strcpy(s1,s2)	p.size = s1.size	s1.size >= s2.size
p = strdup(s)	p.size = s.size	-
p = strncpy(s1, s2, n)	p.size = s1.size	s1.size > n

Figure 6: Some Trusted Library Routines with Implied Constraints and Preconditions

The advantage of providing trusted routines with predefined constraints (rather than including their source code in our analysis) is two-fold. It allows the body of the library routine to use non-affine array accesses or non-type-safe code. Also, we do not need to compute or propagate detailed constraints from the body of the library routine, thus speeding up the analysis.

Finally, to ensure that string routines will not read beyond the size of the array, we always initialize the last character in any array of characters to be null. We added rule **(A5)** to require that the program must not modify the last character, and enforce this rule by excluding the last element in the array size expression used for safety checking.

```
FreeSites(F,P) : set of call sites in F that may call poolfree on pool P directly or indirectly
AllocSites(F,P): set of call sites in F that may call poolalloc on pool P directly or indirectly
PoolsFreed(F)  : set of pool arguments of F that may have a poolfree in F or one of its callees
PoolsAlloced(F): set of pool arguments of F that may have a poolalloc in F or one of its callees

AnalyzeFunction(Function F)
begin
    for (each pool pointer SSA variable P in F)           // formal argument or local variable
        for (each call site FI in FreeSites(F, P))
            for (each call AI in AllocSites(F, P1) where P1 != P)
                if (there exists a path from FI to AI in the Control Flow Graph)
                    Classify (F,P) as ‘‘Case 3’’
            for (each call AI in AllocSites(F, P))
                if (there exists a path from FI to AI in the Control Flow Graph)
                    Classify (F,P) as ‘‘Case 2’’
if !(Case 2 OR Case 3)
    Classify (F,P) as ‘‘Case 1’’
end;

AnalyzeProgram(Program M)
begin
    for (each SCC in CallGraph of M in post-order)
        while (change == true)
            change = false
            for (each function F in the SCC)
                for (each pool pointer variable P in F)     // formal argument or local variable
                    for (each call site CS in F that has P as an argument)
                        for (each function CalledF that can be called at CS)
                            if (CalledF is poolfree for P OR PoolsFreed(CalledF) contains P)
                                if (FreeSites(F,P) does not contain CS)
                                    change = true
                                    add CS to FreeSites(F,P)
                                    if (P is an argument of F)
                                        add P to PoolsFreed(F)
                            if (CalledF is poolalloc on P OR PoolsAlloced(CalledF) contains P)
                                if (AllocSites(F, P) does not contain CS)
                                    change = true
                                    add CS to AllocSites(F,P)
                                    if (P is an argument of F)
                                        add P to PoolsAlloced(F)

        for (each function F in the SCC)
            AnalyzeFunction(F)
end;
```

Figure 4: Algorithm to identify and classify potential memory reuse within and between pools

The pre-conditions and return-value constraints are directly incorporated into our existing analysis for array bounds, described in [15]. We explain the basics of our approach with the help of the example in Figure 7.

```
char A[51];           // last character is set to null
...
k = read(fd, A, 50);  // requires A.size >= 50
if (k > 0) {
  len = strlen(A);    // implies len < A.size
  for (i=0; i < len; i++)
    if (A[i] == '-')
      break;
    ...               // do other stuff with A, i
}
```

Figure 7: Array Usage Example

To prove the safety of any array access we first collect constraints from the index expression and the array size expression by following SSA def-use edges, and collect branch conditions on which those definitions depend by using the control dependence graph. In our example, for array access `A[i]`, the constraints we generate are `(A.size = 51-1)` using the def-use edges from the array declaration (note that the last character is excluded from the size), `(len < A.size && k <= 50)` using the def-use edges and return value constraints on library functions `strlen` and `read`, and `(i < len && k > 0)` from the control dependence graph. Induction variable recognition allows us to generate useful constraints about loop index variables (e.g., `i >= 0`), and (together with the renaming of variables in SSA form) allows us to disregard inconsistent equations like `i = i + 1` for both induction variables and ordinary variables. The complete set of constraints that we generate for this access are `(A.size = 50 && len < A.size && k <= 50 && i < len && k > 0 && i >= 0)`. (Note that the interprocedural constraint propagation is not necessary in this simple example but is essential for most realistic applications in practice.) Finally, we add the illegal array bounds conditions for the reference (`(i < 0 || i >= A.size)` in the example), and then use the Omega library [14] to check if the resulting constraint system is satisfiable. If not (as we have

On all control flow paths,

(A1) The index expression used in an array access must evaluate to a value within the bounds of the array.

(A2) For all dynamically allocated arrays, the size of the array must be a positive expression.

(A3) If an array, `A`, is accessed inside a loop, then

(a) the bounds of the loop must be provably affine transformations of the size of `A` and outer loop index variables or vice versa;

(b) the index expression in the array reference, must be a provably affine transformation of the vector of loop index variables, or an affine transformation of the size of `A`; and

(c) if the index expression in the array reference depends on a symbolic variable `s` which is independent of the loop index variable (i.e., appears in the constant term in the affine representation), then the memory locations accessed by that reference have to be provably *independent* of the value of `s`.

(A4) A set of trusted library routines with specified preconditions may be used, and arguments passed to those routines must satisfy the preconditions.

(A5) The last element of a character array cannot be modified by the program.

Figure 5: Semantic Restrictions on Array Usage

here), the constraints have been proven inconsistent and the array access is safe.

To verify the precondition for the trusted library call `read`, we simply need to check if the negation of the precondition `(A.size >= 50)` along with known constraints on `buf.size` and `count` results in an inconsistent system. Here, `(A.size < 50 && A.size = 50)` trivially results in an inconsistent system. In this manner, we generate and check the preconditions for every trusted library call used by the program.

5. RESULTS

In this section, we address some of the key questions about the effectiveness of our semantic restrictions and compiler techniques used to check memory safety:

1. How much effort is required to convert the existing embedded programs to conform to our semantic restrictions ?
2. Are the pool allocation transformation and heap safety analysis powerful enough to enforce pointer and heap safety statically in different embedded programs?
3. How often do we encounter pools from each of the three categories in these programs?
4. Are the array restrictions flexible enough to permit existing embedded codes (without extensive changes)?
5. Are the semantic restrictions and static analyses for stack safety sufficient for existing embedded codes?

5.1 Methodology and Porting Effort

Our test codes were derived from two embedded application benchmark suites: 13 from MiBench [12] and 4 from MediaBench [19].[7] MiBench consists of embedded codes from a variety of domains including telecommunications, security, networking, etc. MediaBench are predominantly multimedia codes. The program `rasta` use a library called `libsphere` whose source was not available. The experiments for `rasta` assumed that this library is safe and checked the safety of the available source. The benchmarks, their sizes, and our results for each are shown in Table 1.

We found that a few lines of code had to be changed in several benchmarks to conform to our rules, particularly for type safety and array safety. These are shown in the third and fourth columns of Table 1. The two largest changes were for rule **(T3)** in `rasta` and `g721`, which each used a union with a float and an array of four chars to swap the bytes of the float value. We rewrote the code using shift operations and eliminated the union. The other changes for type safety were very small, e.g., initializing local pointer variables before use within their parent function. For the array safety rules, we had to rewrite a few lines of code in 8 programs. The changes were generally minimal and obvious. For instance, in `blowfish` a command line argument was accessed by iterating and checking if the last character was null, which had to be rewritten to use `strlen()` for the loop bound.

Besides requiring very few modifications, the changes themselves were simple and local and in most cases obvious from reading the code or from compiler error messages. Overall, we believe the porting effort to use our compiler for standard C programs is small to negligible.

5.2 Effectiveness of Pointer and Heap Safety Analysis

The *Heap and Pointer Safety* column in Table 1 shows that our compiler was able to enforce safety of heap and pointer usage for *all 17 benchmarks* we studied. About half the benchmarks use no dynamic memory allocation (though they still use pointers). For the other benchmarks, the same column shows the different categories of pools found in each one. The results show that we were able to prove heap safety without increase in memory consumption (i.e., Case 1 or Case 2 pools — no reuse or only self-reuse), for all 13 MiBench benchmarks and 2 of the 4 MediaBench codes.

Only two codes, `rasta` and `epic`, have pools with cross-reuse by other pools (Case 3), which can incur some increase in memory consumption. We believe this is an encouraging result. Both `rasta` and `epic` make extensive use of dynamic memory, yet they contain very few pools that fall under Case 3: just 1 such pool out of a total of 13 pools in `epic` and 5 out of 14 in `rasta`. In fact, 3 of those 5 pools in `rasta` also have self-reuse from the same pool, so that the effect of not freeing memory to other pools is mitigated. We have also observed that some case 3 pools (such as the one in `epic`) could be converted to case 1 or 2 with more sophisticated

[7] Other codes in the benchmarks are not accepted by the current LLVM C front-end, but will be evaluated using a new version of the front-end in the near future.

Benchmark	Lines of Code	Lines of Code Modified for type safety	Lines of Code Modified for array safety	Array Bounds Checker	Heap and Pointer Safety (Case)	Stack Safety
automotive						
basicmath	579	1	3	Yes	Yes	Yes
bitcount	17	5	0	Yes	Yes	Yes
qsort	156	0	1	Yes	Yes	Yes
susan	2122	1	0	No	Yes (Case 1)	Yes
office						
stringsearch	3215	0	3	Yes	Yes	Yes
security						
sha	269	0	1	Yes	Yes	Yes
blowfish	1502	1	5	Yes	Yes	Yes
rijndael	1773	3	6	Yes	Yes	No
network						
dijkstra	348	0	0	No	Yes (Case 2)	Yes
telecomm						
CRC 32	282	0	1	Yes	Yes	Yes
adpcm codes	741	0	0	No	Yes	Yes
FFT	469	0	0	No	Yes (Case 1)	Yes
gsm	6038	0	0	No	Yes (Case 1)	Yes
multimedia						
g721	1622	11	0	No	Yes	Yes
mpeg(decode)	9839	0	0	No	Yes (Case 1)	Yes
epic	3524	4	0	No	Yes (Cases 1,3)	Yes
rasta	7373	13	0	No	Yes (Cases 1,3)	Yes
Totals: 17	39869	39	20	8	17	16

Table 1: Benchmarks, code sizes, and experimental results

compiler analyses where the `pooldestroy` on a pool is moved as close to the last `poolfree` on the pool as possible without compromising safety.

Another interesting use of dynamic memory is seen in `dijkstra`, where a linked list is alive throughout the program and repeatedly allocates and deallocates memory. In a language with explicit regions such as Cyclone [11] or RT-Java, this list would have to go on a garbage collected heap. Finally, there were a number of Case 1 pools, which are amenable to the optimization of turning off individual object frees entirely, effectively performing static garbage collection with no increase in memory usage.

Overall, our results indicate that Case 3 occurs infrequently even in complex embedded codes and typically never occurs at all in the simpler codes. This is strong empirical evidence that our technique is powerful enough to enforce heap safety statically in a broad range of embedded codes.

5.3 Effectiveness of Stack Safety Checks

Our stack safety check ensures that pointers to the stack frame in a function are not accessible after that function returns. The last column of Table 1 shows that only 1 program (`rijndael`) failed this check. This occurred because Data Structure Analysis is flow-insensitive and can yeild false positives. In `rijndael`, a pointer to a local variable is stored in a global but the global is reinitialized by a callee of the function before the function returns. Such cases must be handled by restructuring the program. Overall, these results indicate that stack safety should not be a significant obstacle for static safety checking with our approach.

5.4 Effectiveness of Array Access Checks

Our array bounds checker passed 8 of the 13 benchmarks from MiBench and none from MediaBench, after the few changes described earlier. Interestingly, our tests detected 3 potential array bound violations in the MiBench suite and 2 in MediaBench: one each in `dijkstra` (both the large and small versions) and `blowfish` and two violations in `g721`. All of the errors except the ones in g721 were due to incorrect assumptions on number of command line arguments. The error in g721 was in using a fixed size buffer to copy a file name obtained from a command line argument. This could cause a stack corruption.

The array bounds checking algorithm failed to prove safety for 9 of the codes. Two of these codes used non-affine bit operations on the index variables. 5 other codes use indirect indexing for arrays, e.g., A[B[j]]. One possible solution we aim to explore is to use Ada style subrange types for index expressions, and attempt to prove their safety when *the index values are computed.*

Another two codes use memory locations in the heap to store the size of an array, then load and use this size value in another function, requiring the compiler to prove that the heap location is not modified in between. We believe that this can be handled fairly simply by interprocedural load value numbering.

Overall, safety checking of complex array references remains the most significant obstacle to our goal of 100% static safety checking for a broad class of embedded applications.

5.5 Comparison with Control-C

All the control codes studied in our previous work on Control-C are accepted by our new compiler fully automatically, i.e., do not require the explicit use of single-region operations for dynamic memory management. Perhaps more importantly, the applications with Case 2 and Case 3 pools (Table 1) and many of those with Case 1 pools would be very difficult to implement with the single-region restriction of Control-C. Moreover, since all the programs use command line arguments and most use other strings and I/O library calls, none of them would accepted by the array bounds checks in Control-C. Thus, the new heap analysis and the improved array access checks help to support a much larger class of embedded codes than our previous work, and do so without program annotations.

6. RELATED WORK

The broad approach of our work has been to identify minimal semantic restrictions on imperative programs and to develop new compiler techniques that together permit complete static checking of memory safety, without runtime checks or garbage collection. To our knowledge *no other programing language or compiler system achieves this goal for any non-trivial class of programs.* We believe our results show that we have achieved the goal for a significant subclass of embedded C programs, and the subclass is quite broad if array bounds checks are ignored.

Several alternative approaches have been taken to eliminate specific types of runtime overheads, and we compare our approach with those below.

The Real-Time Specification for Java (RT Java) [4] enables programmers to avoid garbage collection entirely for subsets of the heap by providing three additional types of *MemoryArea*s that are not garbage collected. Runtime checks are required for ensuring safety of references between the different areas. Of these, the *ScopedMemory* type defines nested (i.e., scoped) regions for dynamic allocation. It is much more restrictive and has more runtime overheads than our pools: memory can only be allocated from the current region, it requires the programmer to specify region entry/exit points, and perhaps most importantly, it requires runtime checks to ensure that there are no references from objects in an outer scoped region (or from a different type of memory area) to an inner one [4]. Finally, RT Java also inherits the other runtime checking needs of standard Java such as for arrays, null pointer checks and type coercions.

Real time garbage collection techniques (e.g., see [2] and the references therein) use incremental collection methods to reduce the unpredictability of garbage collection. Such techniques can incur fairly high memory overhead to achieve acceptable real time behavior, up to 2.5 times the actual space consumption of a program in recent work [2].

As an alternative to garbage collection, several recent languages (e.g., RT Java [4], Cyclone [13, 11], and others [9, 5]) have adopted mechanisms for region-based memory management. These languages disallow direct deallocation of items within a region in order to ensure program safety. As discussed in the Introduction, these languages have two key disadvantages relative to our work: (a) they generally require extensive programmer annotations to identify regions; and (b) they provide no mechanisms to free or reuse memory within a region, so that data structures that shrink and grow (with non-nested object life times) must be put into a separate garbage-collected heap or may incur a potentially large increase in memory consumption. (e.g., Cyclone and RT Java both include a separate garbage collected heap.) Automatic region inference [25, 11] can eliminate or mitigate the first but not the second, and has only been successful for type-safe languages without explicit deallocation.

In contrast to these approaches, we infer regions automatically, we use no garbage collection, we permit explicit deallocation of individual data items within regions, and we ensure program safety through a combination of using homogeneous regions and additional static analysis. There are two potential disadvantages in our work, however. We do not prevent certain kinds of errors such as dangling pointer references (this is irrelevant for correct programs). Second, we rely heavily on interprocedural analysis (many of the annotations in Cyclone and other languages are designed to avoid this need), but we retain the benefits of separate compilation by performing all our analysis at link-time (a key advantage of using the LLVM compilation framework [16]).

Boyapati *et al.* [5] present a static type system combining ownership types with region types, to eliminate the runtime checks needed for ensuring safe region deallocation in RT Java. As a region-based language, they have the same differences from our work as discussed above. They provide an additional mechanism based on “sub-regions” of a region for sharing region data safely across threads, using reference counts to reclaim the data. We do not support multi-threaded applications so far.

Linear types and alias types [6, 28, 7, 8] have been used to statically prove memory safety in the presence of explicit deallocation of objects. They achieve this primarily with severe restrictions on aliases in a program, which so far have not proved practical for realistic programs. One of these languages, Vault [7], also uses such a type system (much more successfully) to encode many important correctness requirements for other dynamic resources within an application (e.g., file handles and sockets). It would be very attractive to use Vault’s mechanisms within our programming environment to statically check key correctness requirements of system calls and trusted libraries.

A valuable strategy for compiler-based secure and reliable systems is Proof-Carrying Code (PCC) [21]. The benefit of PCC is that the safety checking compiler (usually a complex, unreliable system) can be untrusted, and only a simple proof checker (which can be made much more reliable) is required within the trusted code base. Fundamentally, PCC does not change what aspects of a program require static analysis and what require runtime checking – that still depends on the language design and compiler capabilities. Thus, PCC is orthogonal to our work, and could be valuable for taking our safety-checking compiler outside the trusted code base.

There has been extensive work on static elimination of array bounds checks (e.g., see [3, 27]), but the goal of that work is generally to eliminate a subset of bounds checks since complete elimination is impossible for standard languages. In contrast, we impose carefully chosen language restrictions to enable compiler analysis to eliminate such checks entirely in conforming programs. Our previous work [15] discusses how the interprocedural bounds checking algorithm presented there compares with related work. Wagner *et al.* have developed a tool for detection of buffer overrun vulnerabilities in general C codes. Their analysis is necessarily imprecise, however, both in terms of generating constraints (flow-insensitive) and solving them, resulting in many false positives. In contrast, we use a more precise context-sensitive analysis and a more rigorous constraint solver.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have described a set of semantic restrictions and compiler techniques that together enable 100% static checking of memory safety for a significant class of type-safe embedded programs. The semantic restrictions are defined on a low-level language-independent type system and instruction set, and implemented in a language-independent, link-time compiler framework.

The key new result in this work is to show how an Automatic Pool Allocation transformation allows us to ensure the safety of dynamic memory management and pointer usage using static checking alone (i.e., without garbage collection or runtime checks on memory operations) and without *any* new syntax. The compiler analysis helps to pinpoint the infrequent case where certain data structures could experience an increase in memory consumption. Our results show that these techniques allow us to check heap and pointer safety for all 17 embedded programs we studied. Our previous techniques for eliminating null pointer checks and for stack safety are also very effective for nearly all these programs, but our current analysis for checking array references can do complete checking for only half the benchmarks we studied.

Overall, we believe that codes certified as safe by our compiler can execute as fast as the those compiled by a native C compiler, while guaranteeing memory safety. Furthermore, we usually require minimal, simple, and completely portable rewriting of existing C programs to make them conform to our restrictions (often improving over the original!).

There are some key steps remaining before we can achieve our long term goal of a secure, low-overhead programming environment based on the techniques above. First, we must explore better language and compiler support for complex array operations. Second, we must provide a robust and flexible runtime environment with mechanisms to enforce correct usage of system calls and runtime libraries. Finally, we must develop an architecture that tolerates bugs in the necessarily complex compiler and runtime system.

8. REFERENCES

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. SIGPLAN '94 Conf. on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[2] D. Bacon, P. Cheng, and V. Rajan. A real-time garbage collector with low overhead and consisitent utilization. In *Proc. 30th ACM Symp. Principles of Programming Languages (POPL03)*, Jan. 2003.

[3] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *SIGPLAN Conf. on Prog. Lang. Design and Implementation*, June 2000.

[4] G. Bollella and J. Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, 2000.

[5] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[6] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 262–275, New York, NY, 1999.

[7] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, Snowbird, UT, June 2001.

[8] M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[9] D. Gay and A. Aiken. Memory management with explicit regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323, Montreal, Canada, June 1998.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Sun Microsystems, 2000.

[11] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, June 2002.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, Dec. 2001.

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

[14] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Computer Science Dept., U. Maryland, College Park, Apr. 1996.

[15] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proc. 2002 Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Grenoble, Oct 2002.

[16] C. Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[17] C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, Jun 2002.

[18] C. Lattner and V. Adve. Data structure analysis: An efficient context-sensitive heap analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.

[19] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[20] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[21] G. C. Necula. Proof-carrying code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, Paris, Jan. 1997.

[22] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proc. 29th ACM Symp. Principles of Programming Languages (POPL02)*, London, Jan. 2002.

[23] L. Sha. Dependable system upgrades. In *Proceedings of IEEE Real Time System Symposium*, 1998.

[24] L. Sha. Using simplicity to control complexity. *IEEE Software*, July/August 2001.

[25] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Trans. Prog. Lang. Sys.*, 20(1), 1998.

[26] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, pages 132(2):109–176, Feb. 1997.

[27] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[28] D. Walker and G. Morrisett. Alias types for recursive data structures. *Lecture Notes in Comp. Sci.*, vol. 2071, 2001.