Memory Safety Without Garbage Collection for Embedded Applications

DINAKAR DHURJATI, SUMANT KOWSHIK, VIKRAM ADVE,
and CHRIS LATTNER
University of Illinois at Urbana-Champaign

Traditional approaches to enforcing memory safety of programs rely heavily on run-time checks of memory accesses and on garbage collection, both of which are unattractive for embedded applications. The goal of our work is to develop advanced compiler techniques for enforcing memory safety with minimal run-time overheads. In this paper, we describe a set of compiler techniques that, together with minor semantic restrictions on C programs and no new syntax, ensure memory safety and provide most of the error-detection capabilities of type-safe languages, without using garbage collection, and with no run-time software checks, (on systems with standard hardware support for memory management). The language permits arbitrary pointer-based data structures, explicit deallocation of dynamically allocated memory, and restricted array operations. One of the key results of this paper is a compiler technique that ensures that dereferencing dangling pointers to freed memory does not violate memory safety, *without annotations, run-time checks, or garbage collection*, and works for arbitrary type-safe C programs. Furthermore, we present a new interprocedural analysis for static array bounds checking under certain assumptions. For a diverse set of embedded C programs, we show that we are able to ensure memory safety of pointer and dynamic memory usage *in all these programs* with no run-time software checks (on systems with standard hardware memory protection), requiring only minor restructuring to conform to simple type restrictions. Static array bounds checking fails for roughly half the programs we study due to complex array references, and these are the only cases where explicit run-time software checks would be needed under our language and system assumptions.

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*

General Terms: Security, Languages

Additional Key Words and Phrases: Embedded systems, compilers, programming languages, static analysis, security, region management, automatic pool allocation

This work has been sponsored by the NSF Embedded Systems program under award CCR-02-09202 and in part by an NSF CAREER award, EIA-0093426 and by ONR, N0004-02-0102.
Authors' address: Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner, Department of Computer Science, 4307A Siebel Center, University of Illinois at Urbana-Champaign, Urbana, IL 61801; email: dhurjati@uiuc.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2005 ACM 1539-9087/05/0200-0073 $5.00

1. INTRODUCTION

Programming environments, including programming languages, compilers, and run-time systems, play a vital role in improving the reliability and efficiency of software. One key class of properties that is provided by many high-level programming languages is that of *program safety*, which guarantees that certain kinds of semantic errors (e.g., type violations or memory access violations) will not be allowed to occur undetected. Such safety guarantees are important for several reasons. First, and most important, they improve software reliability by ensuring that common programming errors are detected and prevented, either at compile-time or at run-time. Second, they ensure better isolation between different components or modules of a software system, which can be particularly important to enable flexible software upgrades (including safety-critical or mission-critical software), as discussed in Section 2. Finally, in some cases, they allow errors to be detected at development time (e.g., via static checking), which is greatly preferable to run-time error detection in safety-critical or mission-critical systems, and in systems operating under tight performance constraints or energy constraints.

Unfortunately, *most embedded software systems today obtain virtually none of these benefits*. Most such systems are programmed in efficient but "unsafe" languages, especially C or C++, rather than languages that provide the safety guarantees described above. There are several key reasons behind these choices. Virtually all safe languages today, for example, Java [Gosling et al. 2000], Modula-3 [Cardelli et al. 1992], Safe-C [Austin et al. 1994], and CCured [Necula et al. 2002], rely on *garbage collection* to ensure that there cannot be references to deallocated heap memory, and use a variety of *run-time software checks before individual memory operations* such as bounds checks for array references, null pointer checks, and type conversion checks. Garbage collection (GC) algorithms suitable for real-time systems introduce significant overheads in both execution time and memory usage [Bacon et al. 2003], while non-real-time GC algorithms introduce unpredictable run-time delays and also incur some increases in average time and space. Region-based languages attempt to achieve safety without GC [Boyapati et al. 2003; DeLine and Fahndrich 2001; Gay and Aiken 1998; Tofte and Talpin 1997] but require significant porting effort and have to fall back on GC to prevent significant increases in memory usage for some data usage patterns [Bollella and Gosling 2000; Jim et al. 2002] (these are discussed in more detail in Section 7). Finally, the overheads of run-time software checks used in safe languages can also be high: languages such as SafeC, CCured, and Cyclone have reported slowdowns ranging from 20% up to 200% for different applications [Austin et al. 1994; Grossman et al. 2002; Necula et al. 2002]. These potential drawbacks have made the use of existing safe languages unattractive for embedded software applications.

The broad outcome of this paper is to show that, *with minor semantic restrictions on C programs and no new syntax, we can provide nearly all the benefits of safe languages, without using garbage collection, and with few or no run-time software checks*. The minimum guarantee we provide is *memory safety*. 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 data area allocated by or for that entity, and (b) it never executes instructions outside the code area created by the compiler and linker within that space. The importance of memory safety for embedded systems is motivated in Section 2. Furthermore, with the exception of dangling pointer references (discussed below), we *detect and prevent all other errors that would be prevented by a language with strong type safety* [Oaks 2001], so that programs also obtain most of the error detection and prevention benefits of type-safe languages. Furthermore, for systems with hardware-supported address space protection and a reserved address range, we achieve this without using any run-time *software* checks (i.e., explicit code before a memory reference), except on complex array references. On other systems, we would also require software null-pointer checks.

These results are based on some novel compiler techniques, combined with several existing techniques. Perhaps the most novel technique developed in this work is a fully automatic compiler strategy that guarantees memory safety even in the presence of "dangling pointer references" to freed heap objects. Our strategy ensures that such references will never lead to a memory access violation, and in fact never allows a memory object to be accessed in a manner that violates its declared type (even through dangling pointers). This technique works for C programs that obey ordinary type rules, requires no new annotations or language mechanisms, and permits explicit memory deallocation. This technique builds on a transformation previously developed by us called automatic pool allocation [Lattner and Adve 2002, 2005], which partitions instances of logical data structures (as identified by the compiler) into distinct pools in the heap, while allowing explicit allocation and deallocation of objects within the data structures. Automatic pool allocation does not itself ensure memory safety, but we show how it can be used to do so in this paper. The solution is important for achieving memory safety without GC because reliably detecting or preventing dangling references automatically (in the presence of explicit memory deallocation) is extremely difficult, either via compile-time analysis or run-time checks [Patil and Fischer 1995; Heine and Lam 2003].

We also present a powerful interprocedural algorithm for propagating constraints on integer variables, which enables us to prove the safety of *affine* array references at compile time using existing integer programming techniques [Kelly et al. 1996; Pugh 1992]. The interprocedural algorithm is important so that array bounds checking can be performed without requiring annotations for function calls and interfaces. We also incorporate support for checking operations on null-terminated strings and for safe usage of a set of trusted run-time functions (e.g., I/O operations and system calls) directly into the same constraint propagation and checking framework used for array bounds checking. Pointer arithmetic is handled using techniques similar to that of checking array accesses.

We combine these new strategies with two existing compiler-based safety checking techniques. One is a memory initialization technique that takes advantage of hardware-supported address space protection to prevent uses of uninitialized pointers, without any software null-pointer checks at run time

(essentially, converting them into a run-time hardware check). The second is traditional escape analysis to detect conservatively the stack locations that are accessible from outside a function to prevent the use of dangling pointers to stack locations.

Some of our safety checking techniques (particularly for null pointer dereferences, dangling pointers to stack locations, and array safety) require minor semantic restrictions, in addition to the ordinary type usage restrictions needed for heap safety. Taken together, these semantic restrictions define a language that is essentially a subset of C (there are no new keywords or syntax) and guarantee that programs obeying these restrictions will obtain the safety guarantees described earlier without incurring any run-time software checks (and without GC). In practice, we find that the complex array references are the most likely to violate these restrictions, whereas the other restrictions are usually met immediately or with minor program changes.

We evaluated the applicability and the memory overhead of our language and compiler analyses for memory safety on a diverse collection of embedded programs from two widely used benchmark suites, MiBench [Guthaus et al. 2001] and MediaBench [Lee et al. 1997], as well as some control and sensor applications. Our results show that we are able to ensure the safety of pointers and dynamic memory usage *in all these programs* without incurring any run-time overhead. This is due to a combination of our technique for preventing null pointer dereferences using a run-time hardware check and the static technique for ensuring safe dereferencing of dangling pointers, both of which work successfully for all the programs. Our compiler analysis identifies specific data structures in three of these programs where our memory management strategy could lead to some potential increase in memory consumption, and we found that in all the three cases the actual increase is not significant. The static technique for checking the lifetimes of stack-allocated data works successfully for 19 of the 20 codes tested. The static array bounds checking analysis, however, is completely successful for only 11 out of the 20 codes. The other codes would require some run-time software checks. We draw some ideas for future language and compiler mechanisms that might succeed for the other programs. Overall, these results show that with the exception of array bounds checks, the set of compiler techniques in this work is able to achieve memory safety without garbage collection and without run-time software checks, for a language that is essentially a "type-safe" subset of C, including programs with complex pointer-based data structures and extensive heap usage. To our knowledge *no other programming language or compiler system achieves this goal for any nontrivial class of programs*.

The rest of this paper is organized as follows. Section 2 describes our goals for compiler-based checking and minimizing run-time overhead, and summarizes the assumptions we make about programs and systems in our current work. Section 3 gives the necessary background information to understand our techniques. Section 4 describes the language restrictions and the compiler techniques for ensuring safety of pointer references, stack safety, and heap safety. Section 5 does the same for array references. Section 6 describes our experiments evaluating the effectiveness of our techniques in supporting different

classes of embedded and control applications. Section 7 compares our work with previous work on providing program safety through compiler and language techniques. Section 8 concludes with a summary of our results and suggests directions for further research.

2. GOALS AND ASSUMPTIONS

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 [Sha 1998, 2001] and sensor networks [Levis and Culler 2002]. 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, that is, to guarantee that an upgraded software module cannot corrupt the code or data of its host application. Moreover, many of these embedded systems must operate under stringent energy, memory, and processing power limitations, and often under hard or soft real-time constraints as well. This reinforces the importance of minimizing the run-time overheads of memory safety.

2.1 Goals

The goal of this work is to enforce memory safety and (wherever possible) detect and prevent type errors and memory access errors under two constraints: (a) permit explicit memory deallocation instead of requiring some form of automatic memory management (e.g., garbage collection or a region-based memory system), and (b) avoid introducing *any* run-time software checks before program operations (e.g., null pointer, array bounds, or type conversion checks). Some run-time support is necessary but we consider it acceptable, specifically, initialization of global or dynamically allocated memory, and some system assumptions for error detection described later.

Achieving the above goals for arbitrary C programs is extremely difficult, and perhaps impossible. First, some of the compiler techniques in this paper are only applicable to programs that follow basic type-safety rules.[1] Second, some language constructs (e.g., unanalyzable array references or pointers to stack-allocated memory that outlive the stack frame) make it impossible to ensure memory safety without run-time software checks. We define a set of restrictions on programs that are sufficient to allow our compiler techniques to be applied, and to eliminate the need for run-time software checks (in one case, our restrictions allow a software check to be converted to an efficient hardware check). To make it as simple as possible to modify existing embedded code to conform to our restrictions, we avoid adding *any* new language mechanisms

[1] In our ongoing work, we are exploring extensions that allow these techniques to be used for arbitrary programs, while requiring only some additional software run-time checks but still avoiding the need for automatic memory management.

or syntax. Instead, we impose usage (i.e., semantic) restrictions that can be defined within the framework of an existing language such as C, and checked by a compiler.

This approach may "fail" in two ways for a given input program. One kind of failure is that we are unable to eliminate some explicit software run-time checks, that is, we fail to meet constraint (b) above. Note that in practice, we could insert software checks for such cases, although we do not do so in this work. Alternatively, we would also consider it a failure if extensive source changes to the input program were required to meet our restrictions (especially, the type safety restrictions needed for our current work). Our experiments in Section 6 evaluate how often these two kinds of failure occur for a wide range of embedded applications.

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) [Lattner and Adve 2004].[2] 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.

2.2 Assumptions of This Work

The system assumptions of the current work are summarized below. The semantic restrictions and other techniques for basic type safety, pointer initialization, stack safety, array safety, and heap safety are described in turn, in the following sections.

First, we make some assumptions about the run-time environment. We assume that certain run-time errors are safe, that is, the run-time system can recover from such errors by killing the applet, thread, or process executing the untrusted code.

We assume a safe run-time 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 run-time error, typically triggered by a page fault handler or by a reserved address range in hardware on systems without virtual memory management. In practice, embedded platforms vary widely in their addressability.

— In standard Linux implementations with 32-bit addressing, the high end of the address space is reserved for the kernel (typically 1 GB out of 4 GB). Our technique to handle null pointer dereferences described in Section 4.2 is most suited to such platforms, where we avoid run-time software checks.
— Smaller embedded platforms with 16-bit and even 8-bit addressing typically do not have a reserved address range. In these cases, null pointer checks must be inserted before every load or store.

[2]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 run-time optimization of programs. See `http://llvm.cs.uiuc.edu`.

Rule (P2) in Section 4.2 requires that the size of any structure does not exceed the size of the reserved address range. In the event that a program contains a structure larger than the size of the reserved address space, the programmer can either restructure the code to use smaller structures or run-time null pointer checks are necessary for any references to the structure.

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 5). 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.

3. BACKGROUND: DS GRAPHS AND AUTOMATIC POOL ALLOCATION

Several of the static program safety analyses described in the following sections rely on a core pair of compiler techniques developed in our group, namely data structure analysis (DSA) [Lattner and Adve 2005] and automatic pool allocation [Lattner and Adve 2005].

Data structure analysis (DSA) is a pointer analysis algorithm that is carefully designed to identify disjoint instances of entire pointer-based data structures and their lifetimes, while remaining fast and scalable enough for large, realistic programs. DSA computes a points-to-graph representation that we call a data structure graph (DS Graph) (see Figure 1). DS graphs provide all of the information used in the rest of this work (including automatic pool allocation). We describe DS graphs first and then briefly discuss the properties of DSA needed to achieve our goals while keeping it efficient and scalable.

A DS graph captures compile-time information about the memory objects created by a program and the pointer relationships between them. A separate DS graph is computed for each function in a program, except that all functions within a strongly connected component of the call graph share a single, common points to graph (we do not try to be context sensitive within such recursive regions). Different nodes within the same graph represent distinct memory objects. Formally, a *DS graph* is a directed multigraph, where the nodes and edges are defined as follows:

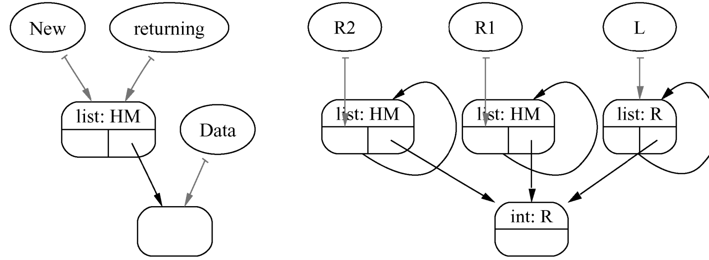
DS Node. A DS node is a 5-tuple $\{\tau, F, M, A, G\}$. τ is some program-defined type, or $\perp$ representing an unknown type. In the analysis, $\perp$ is treated like an unknown-size array of bytes. F is an array of fields, one for each possible field of the type τ. Scalar or array types have a single field. M is a set of memory classes, written as a subset of $\{\mathbf{H}, \mathbf{S}, \mathbf{G}, \mathbf{U}\}$, indicating Heap, Stack, Global, and Unknown memory objects, respectively. A $\mathbf{U}$ node is assigned type $\perp$. Finally, if $\mathbf{G} \in M$, then G is a nonempty set of global variables and functions included in the objects for this node; otherwise, G is empty. Finally, A is a Boolean that is true if the node includes an array object.

DS Edge. A DS edge is a 4-tuple: $\{s, f_s, t, f_t\}$, where s and t are DS nodes, and f_s and f_t are fields of s and t, respectively. Thus, the graph provides a field-sensitive representation of points-to information. A field of a node may have no outgoing DS edge only if the field is known not to contain a pointer

```
struct list { list *Next; int *Data; };
list* createnode(int *Data) {
  list *New = malloc(sizeof(list));
  New->Data = Data;
  return New;
}
void splitclone(list *L, list **R1, list **R2) {

  if (L == 0) { *R1 = *R2 = 0; return; }
  if (some_predicate(L->Data)) {
    *R1 = createnode(L->Data);
    splitclone(L->Next, &(*R1)->Next, R2);
  } else {
    *R2 = createnode(L->Data);
    splitclone(L->Next, R1, &(*R2)->Next);
  }}
```

(a) Fragment of C program manipulating linked lists



(b) DS Graphs for `createnode()` (left) and `splitclone()`.

Fig. 1. Example illustrating data structure graphs.

type, for example, it is a function, floating point, or small integer type, or if $M = \{\mathbf{U}\}$.

Figure 1(b) shows the DS graph computed by our compiler for function `splitclone` of the example in Figure 1(a). Note that each node of type `list` has two fields. The cycles indicate recursive data structures.

The DSA algorithm, which computes DS graphs, is described in Lattner and Adve [2003]. DSA is "fully context sensitive" in the sense that it names heap objects by entire acyclic call paths (which we refer to as "full heap cloning"), and it is field sensitive, that is, it distinguishes distinct pointer fields within a structure. Being fully context sensitive is important because it allows the analysis to distinguish heap objects that may be created, processed, and destroyed by calling common functions. This enables automatic pool allocation (which is based on DS graphs, as described below) to put distinct instances of the same logical data structure into distinct pools in many cases. In the DS graph for `splitclone` in Figure 1, $R1$ and $R2$ point to distinct nodes, indicating that the analysis has proved the two linked lists are completely disjoint. This allows pool allocation to put these two lists in distinct pools, even though they are created in an interleaved fashion and by calling the same function.

To achieve speed and scalability, DSA is flow insensitive and it uses a unification-style analysis, that is, a pointer field at a node has exactly one outgoing DS edge so that all pointer target objects of the pointer are merged into a single graph node. We argue in Lattner and Adve [2003] that the combination of full heap cloning with unification is scalable to very large programs in practice. DSA correctly analyzes non-type-safe programs and incomplete programs, and infers the call graph incrementally as part of the analysis using a new, noniterative technique. The first two properties are essential for real-world use.

DSA actually computes multiple DS graphs for each function. The "complete bottom-up" DS graph for a function incorporates the effects of all functions reachable from the current function (i.e., immediate callees and their callees and so on), including functions called via function pointers [Lattner and Adve 2005]. The final, "top-down" DS graph of a function incorporates the effects of both the callers as well as the callees of a function, so that it captures the full set of memory objects and aliasing relationships from all possible call sites (as well as those due to side effects of callee functions).

We have evaluated DSA experimentally for over 35 C programs, and found that it is both extremely efficient and scales well across a large range of program sizes [Lattner and Adve 2003]. DSA requires about 2–8 s and less than 16 MB of memory for several C programs ranging in size from 60K to 130K lines of code. Empirically, it also scales almost linearly in analysis time for 35 benchmarks spanning 4 orders-of-magnitude of code size. No previous algorithm we know of has demonstrated both speed and scalability with full heap cloning. DSA is compared with previous pointer analyses in more detail in Lattner and Adve [2003].

3.1 The Automatic Pool Allocation Transformation

Given an ordinary imperative program that uses explicit allocation (e.g., `malloc`) and deallocation (e.g., `free`), the automatic pool allocation transformation [Lattner and Adve 2005] rewrites the program to segregate heap objects into multiple pools, by performing allocation and deallocation operations from those pools. The transformation attempts 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. This differs from other automatic region inference algorithms that infer regions primarily for performing automatic memory management [Chin et al. 2004; Tofte and Birkedal 1998], which do not consider data structure relationships in choosing how to partition objects into regions.

We use a pool allocation library with five simple operations: (a) `poolinit (Pool** PP, unsigned size)` allocates and initializes a new pool descriptor for objects of the specified size; (b) `pooldestroy(Pool* PP)` clears the pool descriptor and releases the remaining memory in the pool back to the system heap; (c) `poolalloc (Pool* PP, unsigned nbytes)` allocates a single object or an array of objects in the pool, depending on `nbytes` and the size of the objects in the pool; and (d) `poolfree (Pool* PP, T* ptr)` deallocates an object within the pool by marking its memory as available for reallocation by `poolalloc`. A

```
f() {
  ...
  g(p);
  // p->next is dangling
  p->next->val = ... ;
}
```

```
g(struct s *p) {
  create_10_Node_List(p);
  initialize(p);
  h(p);
  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);
  }
}
```

Fig. 2. Pointer safety and pool allocation example.

pool is created before the first allocation for its data structure instance and is destroyed at a point where there are no accessible references to data in the pool. 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.

To illustrate the pool allocation transformation we use the example in Figure 2. In this example, 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.

The pool allocation transformation operates as follows:

(1) *Identify pools within each data structure*: We traverse the complete bottom-up data structure graph (DSG) of each function to identify heap nodes. Each heap node in this DSG corresponds to objects of a single data type, allocated within the current function or one of its callees. Objects corresponding to this node are allocated in a single pool.
(2) *Identify where to create/destroy pools*: For each procedure, the DSG can be used to identify those DS nodes that are not accessible after the procedure returns (i.e., nodes that are not reachable from globals, formal arguments and return value). For each such node, we insert calls to create and destroy the corresponding pools of memory at the entry and exit of the procedure.[3] 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 3.
(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

[3] 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.

```
f() {
  Pool *PP;
  poolinit(PP, sizeof(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);
  }
}
```

Fig. 3. Example after pool allocation transformation.

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). The transformation uses the call graph constructed by DSA for all interprocedural steps and correctly handles programs with function pointers and recursion. The changes are illustrated by the functions `g()` and `h()` and their invocations in Figure 3.

The result of this transformation for type-safe programs is that all heap-allocated objects are assigned to type-homogeneous pools, nodes in disjoint data structure instances identified by DSA 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 allow data of arbitrary types to be accessed, and could violate memory safety.

4. SAFETY OF POINTER REFERENCES

Enforcing safe pointer usage consists of ensuring basic typing rules, handling uninitialized pointers, stack safety, and heap safety.

4.1 Basic Issues

The first set of restrictions is the typing rules that are summarized 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.

(**T1**) Our type system is the same as that of the C language, but is further restricted by rules **T2** and **T3**.

(**T2**) Casts *to* a pointer type from any other type are disallowed, except certain pointer-to-pointer casts for compatible targets. Permitted casts include casts between pointers to primitive types of the same size or casts from a pointer to a primitive type to a pointer to a primitive type with a smaller size.

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

Rule (**T3**) is similar to rule (**T2**) as unions are implemented using casts. The exceptions to rule (**T2**) are essentially reinterpreting casts for the target numerical value. Note, however, that if the pointer points to an array, the resulting pointer would have no size information and hence any subsequent array index operations would likely be rejected as unsafe by the array bounds checking algorithm (Section 5). Explicit array declarations are just as in C, that is, they need to specify the size of each dimension, except for the first dimension of an array formal parameter. Enforcing the above rules is trivial in LLVM [Lattner and Adve 2004], where all operations are typed and only an explicit `cast` instruction can be used to perform any type conversion.

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 (we ignore array references here, since they are addressed in Section 5): (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 problems must be detected and disallowed at compile-time where possible, and safely detected or tolerated at run time otherwise, without introducing explicit software checks before individual memory references. Below, we examine each of these conditions in turn.

4.2 Uninitialized Pointers

We employ two restrictions, (P1) and (P2) below, in addition to our requirement that the run-time system have a reserved address range, to ensure that an uninitialized pointer (scalar or an element of an aggregate object) is either never dereferenced or results in a safe run-time error.

(**P1**) Every local pointer variable must be initialized before being referenced, that is, *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.

Our compiler prevents errors due to uninitialized pointer values by statically checking that the program honors rules (**P1**) and (**P2**). Rule (**P1**) is motivated

by the following code snippet, disallowed by our language:

```
int a, *p, **pp; pp = &p; print(**pp); p = &a;
```

Here, the address of uninitialized pointer p is taken before it is initialized, thus making **pp potentially unsafe. Such uses are difficult to detect statically (because the use may be in a different function), and even a flow-sensitive interprocedural algorithm is likely to lead to false errors. We prefer to disallow taking the address of an uninitialized pointer. We use a standard global data flow analysis to check rule (**P1**) above that considers only local scalar pointer variables. (Note that interprocedural analysis is not required for identifying uninitialized variables, since any variable needs to be initialized in the calling function before it is passed as an argument).

Detecting uses of uninitialized values for global variables and for pointers within dynamically allocated data (e.g., structure fields or array elements) is difficult at compile time. Type-safe language implementations usually initialize pointer fields in aggregate objects to null and use run-time null pointer checks to detect uses of uninitialized values. In order to avoid performing such checks explicitly in software, we initialize all uninitialized global scalar pointers, and all pointer fields in globals and dynamically allocated data structures at allocation time, to point to the base of the reserved address range. This is enabled by our typing rules, which ensure that the type of each dynamically allocated object is known statically. Pointer fields in stack-allocated variables of aggregate types are also initialized to the same value. This includes arrays of pointers in the aggregate type which are initialized in a loop only once at allocation time. 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[4] cannot exceed the size of the reserved address range. With this rule, the above initialization ensures that the effective address for the load or store of any scalar variable or structure field using an uninitialized pointer (e.g., p->X, where p is uninitialized) will fall within the reserved address range, thus triggering a safe run-time error. If a reserved address range is unavailable or the structure size restriction above is unacceptable, then explicit software checks for null pointer references would be required.

4.3 Stack Safety

The second way in which pointer usage can lead to unsafe memory behavior (problem (b) in Section 4.1) is when a pointer into a stack frame of a function is live after the lifetime of the function. This potentially arises when the address of a local variable 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 disallow only placing the address of a stack location in any heap location or global variable, or returning it directly from a function (rule **P3** below). Microsoft CLR's type system [Gordon and Syme 2001] has exactly this restriction.

[4]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.

```
DSG(F)    : Bottom-up data structure graph for function F
ReachableNodes(N, F): DS Nodes reachable from Node N in DSG(F)

for (each function F in program M)
   for (each DSNode N in DSG(F))
       if (N is pointed to by an argument or return value of F or global )
           for (each DS node N' in ReachableNodes(N, F))
              if (N' contains an 'S' (stack) flag)
                  Report ''Rule P3 violated by N'''
```

Fig. 4. Stack safety algorithm.

(**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.

Rule **P3** can 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. Note that this is equivalent to traditional escape analysis for detecting upwards-escaping objects. This algorithm is shown in detail in Figure 4.

4.4 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. We use our running example in Figure 2 to illustrate the challenges. Note the use of dangling pointer in function `f` after `g` returns. In such code, it is extremely difficult for any compiler to identify statically 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 region-based languages) 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 basic principle underlying our approach is the following: (Type homogeneity principle) *If a freed memory block 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 usage in the source—we only 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 run-time overhead. Programs with dangling pointer errors will execute safely but we will not detect such errors for these programs.

One objection to achieving safety via the above principle is that it can make it *more difficult* for programmers to detect dangling pointer errors because such errors would not produce any type violations. During debugging, however, the above principle *need not be used*. In fact, the pool allocation library and run-time system can use many run-time techniques to assist in detecting dangling

pointer errors. During production runs, on the other hand, we believe that the principle *is* appropriate to use and its benefits greatly outweigh any possible loss in error detection. During production runs of embedded software or system software, there is little benefit in halting execution as soon as a potential memory corruption occurs (in fact, many memory corruption errors may not lead to significant failures, so halting execution immediately could be premature). The benefits of ensuring memory safety despite such errors (while avoiding the overheads of garbage collection) are much more significant for such software.

Using the above principle directly, one naïve but impractical *and incorrect* 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. More importantly, the simple solution is incorrect because it would allow errors that make the results of the compiler's pointer analysis invalid, and therefore invalidate any static checks that use pointer analysis (including our stack safety, and array bounds checking algorithms). This problem is explained in the next section.

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.

4.4.1 *Exploiting Pool Allocation for Heap Safety.* The basic principle of type homogeneity mentioned earlier can be applied to ensure program safety after the pool 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 run-time library so that memory of a pool is not released to the system heap except by `pooldestroy`. With this change, any reference via a dangling pointer to a pool object will be guaranteed to reference either the original object or a new object of the same type and alignment as the original, and *belonging to the same pool*. This ensures that the basic principle described above is satisfied.

An important but subtle point is that (stated informally), it is essential that a dangling pointer can only refer to objects from the same pool (DS node) as the original object in order that the results of the pointer analysis (DSA) are valid. This is because the results of DSA indicate that a pointer to a DS node can refer to *any object represented by that node*, but not objects represented by other nodes. If a dangling pointer could refer to an object that logically belongs to some other DS node (but at the same memory address as the original object), then this result of DSA would no longer be valid. More formally, after the sequence `free(q);...; p = malloc(...)`, if `q` is still usable (i.e., it is a dangling pointer), then the modified pool run-time library guarantees that the memory of `*q` will be reused for `*p` *only if* the old object `*q` and the new object `*p` were allocated

using the same pool descriptor. This in turn implies that q and p must have been inferred to point to the same DS node, since we use a distinct pool descriptor for each DS node. Thus, the DS graph correctly indicates that q and p may point to the same object. In other words, our pointer analysis results are valid for any execution of the program, including executions that may cause a dangling pointer such as q to be dereferenced.

Note that the naïve solution of using one pool per static type would *not* ensure correctness of pointer analysis. This is because, in the above example, p could reuse the memory of q even though p and q point to two different DS nodes holding the same data type. Thus, *p and *q would be aliased in this execution even though DSA claimed they were not. Similarly, our solution based on automatic pool allocation ensures that the results of DSA are correct but if some static analysis (e.g., array bounds checking) used a more precise pointer analysis than DSA, the results of such a pointer analysis may be illegal for some executions due to dangling pointer references. In particular, the pointer analysis results used for any static analysis pass must be no more precise than the pointer analysis used by automatic pool allocation to segregate memory into pools.

4.4.2 *Detecting Potential Increases in Memory Consumption.* The second key issue is memory consumption. The change to the pool allocation run-time library above prevents reuse of memory between two simultaneously live pools. This can have the same disadvantage as the naïve type-based pools—the memory requirement of the program could increase. 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 one of 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 5(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 5(b): after the first poolfree on p1 there

```
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);
  freeSomeItems(p1,t);
  addItems(p1,t);  // self-reuse
  addItems(p2,t);  // cross-reuse
}
freeTree(p1,t);
poolDestroy(p1);
```

(c) Self- and cross-reuse (case 3)

Fig. 5. Example illustrating three types of reuse behavior for a pool `p1`.

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 5(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, the programmer would first analyze or profile the memory consumption of the code, focusing on data structures assigned to case 3 pools identified by our classification algorithm. Typically the programmer has the following choices:

(a) Often, the increase in memory with case 3 pools is acceptable. These are cases where there is limited wasted memory from pools with overlapping lifetimes, in spite of not freeing memory back to the system (possibly due to a lot of pool memory self-reuse).

(b) In some situations, the source code of the program could be restructured to avoid case 3 pools. For instance, since our calls to `poolinit` and `pooldestroy` are at the entries and exits of functions, enclosing the use of a data structure from the point it is first used till its last use within a function potentially moves the `pooldestroy` for the pool earlier in the program. However, case 3 pools are sometimes unavoidable if there are long-lived data structures with overlapping lifetimes.

    Furthermore, standard software engineering practices tend to minimize the number of case 3 pools. Examples include separating long-lived and short-lived data into distinct data structure instances, avoiding long-lived pointers to short-lived data, and modular program design (especially confining data structure instances within functions). These observations are supported by our experimental results, which show that case 3 pools occur in few of our benchmarks, and the increase in memory consumption is small.

Note that the pool in our running example of Figure 3 has only self-reuse, and we can guarantee memory safety without any increase in memory consumption. Our experiments in Section 6 have produced very few instances of case 3; they

occurred only in 3 out of the 20 embedded and control programs we examined, and none had significant increase in memory consumption due to the change to the pool run-time library.

4.4.3 *Compiler Algorithm for Categorizing Pools.* We have developed a compiler analysis to categorize pools into the three cases described above. This algorithm is run after the automatic pool allocation transformation as shown in Figure 9 and identifies to which group each pool belongs. For this static analysis, each call to `poolinit()` is a distinct pool. When analyzing a particular function, each distinct pool descriptor (which may be a formal argument or a call to `poolinit()`) is treated as a potentially distinct pool.

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 declared within a function identifies a distinct pool.

The algorithm for identifying and categorizing reuse within and across pools is shown in Figure 6. 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` on a pool to the `pooldestroy` calls on that pool, looking for all calls to `poolalloc` that appear on such a path. This is shown as routine `AnalyzeFunction` in Figure 6. `AnalyzeFunction` contains an iterative forward dataflow algorithm, which, for each program point, computes the set of all pools that are freed but not destroyed along some path to the point. For each basic block `BB`, the sets `BBfreesBefore(BB)` and `BBfreesAfter(BB)` represent these sets at the entry and exit of the block. The set `BBfreesBefore(BB)` is simply a union of the sets `BBfreesAfter(p)` for all predecessor blocks `p` of `BB`. A pool in the set `BBfreesBefore(BB)` is propagated to `BBfreesAfter(BB)` unless there is a call to `pooldestroy` on that pool in `BB`. Each iteration is a linear-time traversal of the basic blocks, and we have found that there are only a small constant number of iterations in practice (as expected because this dataflow problem has the acyclic propagation property [Aho et al. 1986]). Every `poolalloc` call is then analyzed by checking for calls to `poolfree` on undestroyed pools on any path preceding it. This is computed using `BBfreesBefore` for the basic block corresponding to the `poolalloc` call and the set of pools freed but not destroyed in the basic block before the `poolalloc` call. Pools are categorized based on the instances of `poolfree` (if any) found on such paths.

```
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
// Analyze direct and indirect calls to poolalloc, poolfree and pooldestroy and classifies pools in a function
AnalyzeFunction(Function F)
begin
    BBfreesBefore(BB) : set of pools freed but not destroyed on some path to the beginning of basic block BB
    BBfreesAfter(BB)  : set of pools freed but not destroyed on some path to the end of basic block BB
    BBdestroys(BB)    : set of pools destroyed in basic block BB
    for (each basic block BB in F)
        initialize BBfreesAfter(BB) with pools freed in BB
        initialize BBdestroys(BB) with pools destroyed in BB
        BBfreesAfter(BB) = BBfreesAfter(BB) - BBdestroys(BB)
    while (change) // forward propagate frees on pools, kill free upon destroy
        for (each basic block BB in F in a reverse post-order traversal of the CFG)
            BBfreesBefore(BB) = Union of BBfreesAfter(pred(BB)) for all predecessors of BB
            BBfreesAfter(BB) =  BBfreesAfter(BB) union (BBfreesBefore(BB)) - BBdestroys(BB)
            Recompute change
    // Classify pools as Case 1, 2 or 3
    for (each call site AI in AllocSites(F,P))
        AIBB: basic block corresponding to AI
        BBdestroysBeforeAI = set of pools destroyed before AI in AIBB
        BBfreesBeforeAI =  BBfreesBefore(AIBB) UNION (set of pools freed preceding AI in AIBB)
        for (Pool P1 in (BBfreesBeforeAI - BBdestroysBeforeAI))
            if (P1 == P) Add (F, P1) to ``Case 2 Pools''
            else         Add (F, P1) to ``Case 3 Pools''
    if (!(Case 2 or Case 3))
        Add (F, P) to ``Case 1 Pools''
end;
// Propagate calls to poolalloc and poolfree interprocedurally and analyze pools in each function
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)
                // Compute AllocSites(F,P), FreeSites(F,P), PoolsFreed(F) and PoolsAlloced(F)
                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;
```

Fig. 6. Algorithm to identify and classify potential memory reuse within and between pools.

Consider next an input program without recursion. The algorithm then makes a bottom-up traversal of the call 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 assume that `pooldestroy` on a pool is only called at the function in which the pool is created

and hence we do not need to propagate these calls interprocedurally. 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 nonrecursive programs uniformly, we actually perform the bottom-up traversal on the strongly connected components (SCCs) 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.

5. ARRAY RESTRICTIONS

In general, array operations are one of the most expensive to check for memory safety at run time. We identify some restrictions on the language (as few as possible, given the state of the art of static program analysis) that can allow us to verify statically the safety of all array accesses in a program.

5.1 Language Design

From the viewpoint of language design for static safety checking, one of the fundamental limits in static program analysis lies in the analysis of constraints on symbolic integer expressions. For ensuring safety, the compiler must prove (symbolically) that the index expressions in an array reference lie within the corresponding array bounds on all possible execution paths. For each index expression, this can be formulated as an integer constraint system with equalities, inequalities, and logical operators used to represent the computation and control-flow statements of the program. Unfortunately, satisfiability of integer constraints with multiplication of symbolic variables is undecidable. A broad, decidable class of symbolic integer constraints is *Presburger arithmetic*, which allows addition, subtraction, multiplication by constants, and the logical connectives $\vee$, $\wedge$, $\neg$, $\exists$, and $\forall$. (For example, the Omega library [Kelly et al. 1996] provides an efficient implementation that has been used for solving such integer programming problems in compiler research.) Exploiting static analysis based on Presburger arithmetic requires that programs only use linear expressions with (known) constant coefficients for all computations that determine the set of values accessed by an array index expression.

With this intuition, we derive a set of language rules for array usage. First, recall the definition of an affine transformation. Let $F : R^n \rightarrow R$. Then a transformation F is said to be *affine* if $F(\vec{p}) = C\vec{p} + \vec{q}$, where C is any linear transformation, and $\vec{q}$ contains only constants or known symbolic variables independent of $\vec{p}$. In the following, we assume affine transformations with known constant integer coefficients (C).

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**) For every reference of an array `A`, either the index expression in the array reference must be a provably affine transformation of the size of `A` or the following must hold:
  (a) the array reference has to be inside a loop;
  (b) the index expression in the array reference must be a provably affine transformation of the vector of index variables of enclosing loops;
  (c) the bounds of the enclosing loops must be provably affine transformations of the size of `A` and outer loop index variables or vice versa; and
  (d) if the index expression in the array reference depends on a symbolic variable `s` which is independent of a 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`.

A1 by itself guarantees safe array accesses, but the compiler can check that a program satisfies A1 if the additional language rules A2–A3 are obeyed.

The length of an array can be any nonnegative expression. Arrays can also be passed as formal parameters and be returned as return values (using pointers, just as in C), relying on interprocedural analyses during compilation to propagate the array sizes.

Note the use of the phrase "provably affine" in the rules, indicating that the ability of the compiler to prove a transformation is affine plays a role in accepting or rejecting a program. Pointers to arrays, if loaded from memory locations (some multilevel pointers), are typically hard to reason about and we cannot prove them as affine in our system. A more sophisticated compiler might be able to prove more array accesses as affine. In this work, we present a technique that uses interprocedural analysis to prove that a large number of array accesses, which do not involve pointers loaded from heaps, actually use only affine transformations.

Rule A3(d) requires some explanation. A simpler alternative would be to restrict the affine expressions to use only known constants even in the second term ($\vec{q}$), but this is unnecessarily restrictive. For example, a loop could run K to $N + K - 1$ and an index expression within the loop could be of the form $A[i - K]$, where K is some (unknown) loop-invariant value. This array access is easy to prove safe, but would be disallowed under the simpler rule. Instead, A3(d) allows a variable such as K to appear as long as the specific value of K does not affect which array locations are accessed. Thus, in the example, the array locations accessed in the loop are $A[0..N-1]$, regardless of the value of K.

Array accesses and pointer arithmetic are handled uniformly in our code representation. We lower all array accesses to pointer arithmetic operations

Table I. Some Trusted Library Routines with Implied Constraints and Preconditions

Library Call	Return Value Constraints	Safety Pre-conditions
`n = read(fd, buf, count)`	`n <= count`	`buf.size >= count`
`n = puts(s)`	–	–
`p = memcpy(p1, p2, n)`	`p.size = p1.size`	`p1.size >= n`
`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`

(followed by memory accesses) and work on the resulting program. This means that we enforce our semantic restrictions on arrays as well as other legal pointer arithmetic operations uniformly.

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 the following rule **(A4)** to allow trusted string and I/O library routines that make use of arrays:

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

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 (postconditions) relating the output of a routine to its inputs, which can 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)` (as explained later in (**A5**), for a character buffer `A`, `A.size` represents the allocated size - 1) and the constraint on the return value `(n <= count)` since `read` can read only up to `count` bytes. Some trusted library calls and the corresponding constraints are listed in Table I.

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 nonaffine 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 also added the following rule:

(**A5**) The last element of a character array cannot be accessed by the program (trusted library calls like strlen can access it).

(A5) requires that the program must not modify the last character, and we enforce this rule by excluding the last element in the array size expression.

5.2 Compiler Implementation

Compiler checking for safe array usage requires four steps:

—generating constraints from each procedure,
—interprocedural propagation of constraints,
—verifying whether each array access is safe, and
—verifying each safety precondition is true.

5.2.1 *Generating the Constraints.* We generate a set of constraints for each array access in a program, including only those constraints that affect the array access. The algorithm is described in Figure 7. We use a flow-insensitive algorithm that exploits the SSA representation in LLVM. The LLVM instruction set distinguishes registers (which are in SSA form) and memory, and it allocates to registers all local scalar variables (including pointer variables) whose address is not taken. Global variables, heap-allocated data, and address-taken local variables are kept in memory and are not in SSA form. All instruction operands are SSA registers, and memory locations are accessed only via load and store instructions. To get constraints for an array access, `A[i][j]`, we traverse def-use chains backwards from the definitions of the SSA variables holding `&A`, `i`, and `j`, respectively. These constraints are simply inequalities on integer SSA variables that can be inferred from the program statements. For most program statements, generating the constraints is straightforward. For example, from a simple statement like `i = (x + z) * 5`, we would generate an affine constraint $i = 5x + 5z$ (our examples use C syntax, although such a statement would internally require three LLVM instructions and two temporaries, not shown here). No constraints are generated for any nonaffine expression, including a load instruction. Not generating a constraint for a variable makes the variable unconstrained, and the safety checker will treat the array access as unsafe, unless the variable is irrelevant. We recursively traverse the def-use chains for each variable (e.g., x and z), stopping only if we encounter a nonaffine operation, a formal argument, a return value from a call, or an instruction whose constraints have already been computed and cached. We cache the final constraints on each instruction we traverse so that they can be reused.

We explain the basics of our approach with the help of the example in Figure 8.

For array access `A[i]` in Figure 8, the constraints we generate are a conjunction of

—`(A.size = 51-1)` generated using the def-use edges from the array declaration with the last character excluded from the size because of A5.
—`(len <= A.size && k <= 50)` generated using the def-use edges and return value constraints on library functions `strlen` and `read`.
—`(i < len && k > 0)` generated from the control dependence graph using the `ControlDependentConditions` procedure in the algorithm.

```
Constraints ::=   AffineConstraint  // affine equality or inequality interms of program variables
                |  Constraints && Constraints  // conjunction of two constraints
                |  Constraints || Constraints  //disjunction of two constraints.

//Helper Functions
isSatisfiable(Constraints c) : returns true if c is satisfiable, false if not. //uses Omega
ControlDependentConditions(var v) : returns a list of Conditions on which v is control dependent.
def(v) : definition of the variable v
VarsUsed(Variable v) : returns a list of variables used in the def of v
CollectConstraintsAtAllCallSitesOfThisFunction() : merges constraints from all call sites of this function
CollectConstraintsOnReturnValue(Function f) : returns the constraints on return value of f
                                              in terms of its arguments (if affine).
stepfn(v) : if v is an induction variable in a loop, it returns the step value of the loop.

// The following tries to check if array access A[i] is safe
AnalyzeArrayAccess(A, i)
begin
  Constraints c = generateConstraints(A);
  c = c && generateConstraints(i); //We merge the constraints of A and i
  c = c && ((A.size < 0) || (i >= A.size))
  if (isSatisfiable(c)) return false //Access is unsafe
  else return true //Access is safe
end

//The following generates constraints for SSA variable v
generateConstraints(Variable v)
begin
  Constraints c;
  if v is in cache return constraints from cache;
  //First check if the definition of v is an affine expression
  if def(v) is affine arithmeticOperation
       c = def(v); //generate constraints for simple arithmetic operations
       c = c && generateConstraints(VarsUsed(def(v)));
  //check def(v) is a malloc or alloca of an array
  else if def(v) is (non char) array allocation of size d
       c = (v.size = d)
  else if def(v) is char array allocation of size d
       c = (v.size = d - 1)   //This is because of A5
  else if def(v) is formal argument
       c = CollectConstraintsAtAllCallSitesOfThisFunction();
  else if def(v) is return value of a function call
      c = CollectConstraintsOnReturnValue(callee(def(v));
      c = c && generateConstraints(VarsUsed(args(def(v))))
  else if def(v) is a PHI(x1,x2) and def(v) is induction variable of a loop
      and x2 is coming through backward edge of the loop
      if stepfn(def(v)) is positive   c = c && ((v >= x1) || (v <= x2))
      else if stepfn(def(v)) is negative c = c && ((v <= x1) || (v >= x2))
  //We now add  the control dependent conditions and their definitions
  ConditionList = ControlDependentConditions(v);
  foreach (Condition k in ConditionList)
      c = c && k && generateConstraints(VarsUsed(k));
  store constraints of v as c in cache
  return c;
end
```

Fig. 7. Algorithm for array bounds checking.

For an SSA ϕ node, $x_3 = \phi(x_1, x_2)$, we check if it represents an induction variable, using an existing induction variable analysis [Bachmann et al. 1994; Birch 2002]. If the ϕ node is not an induction variable, then we simply ignore the constraints on the ϕ node since this cannot lead to affine constraints. If the ϕ is an induction variable, we know it merges values from a back edge and a forward edge. If the step function of the variable is positive, we add the constraint $((x_3 \geq x_1) \vee (x_3 \leq x_2))$, where x_1 comes from a forward edge and x_2 comes from a backward edge. If it is negative, we add the constraint $(x_3 \leq x_1) \vee$

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

Fig. 8. Array usage example.

($x_3 \geq x_2$). An induction variable with an unknown step function cannot produce affine constraints and will simply be ignored.

Overall, induction variable recognition allows us to generate useful constraints about index variables (e.g., `i >= 0`) and (together with the renaming of variables in SSA form) avoids generating inconsistent equalities like `i = i + 1` for both induction variables and ordinary variables.

The complete set of constraints we generate for the example reference is `(A.size = 50 && len <= A.size && k <= 50 && i < len && k > 0 && i >= 0)`.

5.2.2 *Interprocedural Propagation.* In many cases, size expressions for an array or constraints on variables used in index expressions must be propagated interprocedurally. For this purpose we have developed an algorithm for interprocedural constraint propagation. The interprocedural algorithm consists of two passes on the call graph. First, a bottom-up pass gets constraints on return values in terms of procedure arguments. A top-down pass then merges constraints on arguments coming in to a procedure from different call sites and then tries to prove safety for all array accesses and safety preconditions in that procedure. Our current implementation cannot prove the safety of accesses which depend on recursive functions and hence simply rejects them. Our experiments have shown that array accesses that depend on recursive functions are very rarely provably affine.

The algorithm has a worst-case exponential time complexity. In practice, however, we have found that, for most embedded applications, a simple heuristic like collecting all the constraints for each of the different arrays passed to the procedure and then merging and simplifying them removes many redundant constraints and greatly increases efficiency.

Our static checking algorithm assumes that there will be no overflows and underflows in the integer arithmetic involved in index or array size computations. Statically verifying this is extremely difficult and furthermore, once verified, does not allow us to perform many traditional compiler optimizations that reorder computations (unless we also verify that there is no overflow/underflow for any possible reordering of the computations). Fortunately, many modern processors, though not all, have the ability to raise exceptions on overflow or underflow on arithmetic operations. In the embedded world, most processors

derived from the MIPS instruction set (e.g., MIPS32, MIPS64, R4300i) have such an ability. Among the general-purpose processors, DEC Alpha, VAX, and IBM/S 360 descendants provide such feature. The x86 and sparc architectures do not automatically raise hardware exceptions upon overflow/underflow, but set some flags in condition code register. However, they do provide special "trap on overflow" instructions like *INTV* in x86 and *tcc* in sparc, which when inserted after an arithmetic operation would check the overflow/underflow flag and raise an exception if the flag is set. To guarantee safety on these processors, we would need to insert the corresponding "trap on overflow" instruction after every arithmetic operation that affects an array access.

5.2.3 *Checking for Array-Bound Violations.* We use the Omega integer set library [Kelly et al. 1996] to test each array index expression for safety. Once we generate constraints for an array reference, we add conditions representing array bounds violations for the reference (such as `(i < 0 || i >= A.size)` in the earlier example). We then use the Omega library to check if the resulting constraint system is satisfiable. If the system is not satisfiable (as we have here), the constraints have been proven inconsistent and the array access is safe. Otherwise, the access is potentially unsafe and we reject the program.

5.2.4 *Checking Safety Preconditions.* To verify the preconditions for each trusted library call (e.g., the call to `read` above), we simply need to check if the negation of the precondition (`(A.size >= 50)`) along with known constraints on variables in the argument expressions (`buf.size` and `count`) result in an inconsistent system. Here, `(A.size < 50 && A.size = 50)` is trivially inconsistent. In this manner, we generate and check the preconditions for every trusted library call used by the program.

6. 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 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) How much does the heap safety transformation affect the execution time and the memory usages of the programs?
(5) Are the semantic restrictions and static analyses for stack safety sufficient for existing embedded programs?
(6) Are the array restrictions and bounds-checking algorithm flexible enough to permit existing embedded programs (without extensive changes)?

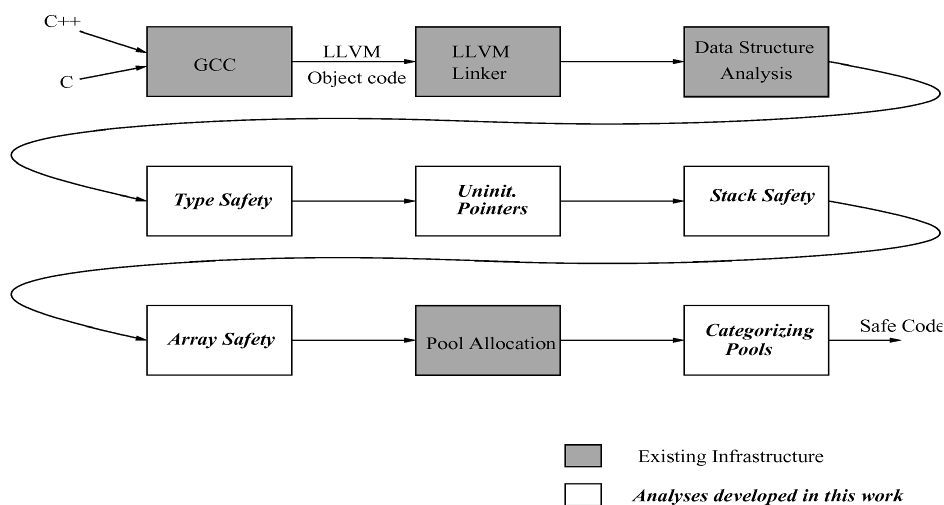


Fig. 9. Implementation.

6.1 Implementation

We have implemented a safety-checking compiler that includes all the compiler techniques described in this paper, using the LLVM compiler infrastructure. Figure 9 is a high-level block diagram showing the sequence of steps we use to enforce safety. Previously existing compiler components are shown by shaded boxes and the rest are new components developed for this work. We have also modified our run-time pool allocation library so it does not release free memory in a pool back to the system heap until the pool is destroyed.

6.2 Methodology and Porting Effort

Our test programs were derived from two embedded application benchmark suites: 13 from MiBench [Guthaus et al. 2001] and 4 from MediaBench [Lee et al. 1997] (of the other programs in MediaBench, two fail pool allocation and one is not accepted by the current LLVM C frontend), two classes of experimental control codes, and sensor network applications. MiBench consists of embedded programs from a variety of domains including telecommunications, security, networking, and so on. MediaBench is predominantly multimedia programs. In addition, we tested a set of PID controllers for an inverted pendulum running on the Simplex real-time architecture [Sha 1998], LQR state space controllers for the Pendubot experiment from the controls laboratory at the University of Illinois, and real-time sensor applications in sensor networks running on TinyOS [Hill et al. 2000]. We believe that these programs cover a wide variety of embedded applications used in practice. The applications that our LLVM C frontend or the pool allocation pass currently refuse to compile are similar to the ones that we report here with respect to code size (except ghostscript) and in usage of dynamic memory, and we do not expect the results to change qualitatively for the remaining benchmarks.

The program `rasta` used a library called `libsphere` whose source was not available. The experiments for `rasta` assumed that this library is safe and

Table II. Benchmarks, Code Sizes, and Analysis Results

Benchmark	Lines of Code	Lines of Code Modified for Type Safety	Lines of Code Modified for Array Safety	Heap and Pointer Safety (case)	Stack Safety	Array Safety
Control						
Pendulum	300 (Average)	0	0	Yes	Yes	Yes
Pendubot	1,300 (Average)	0	31	Yes (case 1)	Yes	Yes
TinyOS apps	300 (Average)	0	0	Yes	Yes	Yes
Automotive						
basicmath	579	1	3	Yes	Yes	Yes
bitcount	17	5	0	Yes	Yes	Yes
qsort	156	0	1	Yes	Yes	Yes
susan	2,122	1	0	Yes (case 1)	Yes	No
Office						
stringsearch	3,215	0	3	Yes	Yes	Yes
Security						
sha	269	0	1	Yes	Yes	Yes
blowfish	1,502	1	5	Yes	Yes	Yes
rijndael	1,773	3	6	Yes	No	Yes
Network						
dijkstra	348	0	0	Yes (case 2)	Yes	No
Telecomm						
CRC 32	282	0	1	Yes	Yes	Yes
adpcm codes	741	0	0	Yes	Yes	No
FFT	469	0	0	Yes (case 1)	Yes	No
gsm	6,038	0	0	Yes (cases 1,2)	Yes	No
Multimedia						
g721	1,622	11	0	Yes	Yes	No
mpeg(decode)	9,839	0	0	Yes (cases 1,3)	Yes	No
epic	3,524	7	0	Yes (cases 1,3)	Yes	No
rasta	7,373	25	0	Yes (cases 1,3)	Yes	No
Totals: 20	41,769	70	53	20	19	11

checked the safety of the available source. Also, for each of the programs above, we designate library and system calls (file reading and writing routines, for instance), whose source is unavailable, as being trusted. This is safe since we manually enforce that each of these programs is linked only to trusted libraries.

The benchmarks, their sizes, and our results for each are shown in Table II.

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 II. The largest changes were for rule (**T3**) in `rasta`, `Pendubot`, and `g721`. All the three programs used unions with incompatible types; `rasta` had a union with a float and an array of four chars to swap the bytes of the float value, and `g721` did the same for an unsigned int. We rewrote the code using shift operations and eliminated the union. The benchmark `epic` used a wrapper around malloc to check the return value of malloc and to exit the program if it were null. This resulted in casts from `char*` to a pointer to the type being allocated after each call to this wrapper function.

We replaced the wrapper function with a plain malloc call in order to prevent these casts. The return value check, however, is preserved since these calls to malloc are converted into `poolalloc`, and the return value of malloc is checked by the `poolalloc` library function. The other changes for type safety were very small. For instance, we needed to initialize local pointer variables before use within their parent function. Also, the `fread` and `fwrite` system calls take a `char*` value as their first argument, leading to a cast from an arbitrary pointer to a `char*` before it is passed as argument. This violates rule (**T2**) and also prevents pool allocation from being applied to the object being passed in. We have defined separate trusted wrapper functions around `fread` and `fwrite` for each primitive nonpointer type, and we changed the source code to use the appropriate wrapper functions. (Programs that read nonprimitive data from a file would be rejected.)

For the array safety rules, we had to rewrite a few lines of code in eight 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 and using an induction variable in the while loop which depends on the `strlen()`. In another case (search_string), an array of strings were accessed in a while loop with the index variable unrelated to the bounds. We had to rewrite the code to make the access obey our language rules described in Section 5.

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.

6.3 Effectiveness of Pointer and Heap Safety Analysis

The *Heap and Pointer Safety* column in Table II shows that our compiler was able to enforce safety of heap and pointer usage for *all 20 programs* we studied. More precisely, the DSA and pool allocation techniques together are sufficiently precise to partition the programs heap data into type homogeneous pools (after the few changes to these programs we made to ensure that the programs pass our type safety requirements). 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, 1 of the 4 MediaBench programs, and all the control programs.

Only three programs, `mpeg2decode`, `rasta`, and `epic`, have pools with cross-reuse by other pools (case 3). In practice, our experimental results (see Table III and Section 6.3.1) have shown that these do not result in a significant increase in memory consumption (see Table III). The three programs, `mpeg2decode`, `rasta`, and `epic` make extensive use of dynamic memory, yet they contain very few pools that fall under case 3: just 4 of the 8 pools in `mpeg2decode`, 3 of 12 in

Table III. Execution Time and Memory Usage for Heap Safety Approach

Benchmark	Execution Time (s)			Memory Usage (bytes)				
	Orig Time	Heap Safety Time	Exec Ratio	Orig Mem Usage	Pool Alloc Mem. Usage	Mem Ratio 1	Pool Alloc. + Safety Restriction	Mem. Ratio 2
Automotive								
basicmath	1.667	1.672	1.00	16,384	16,384	1	16,384	1
bitcount	0.710	0.727	1.02	16,384	16,384	1	16,384	1
qsort	0.405	0.404	1.00	24,576	24,576	1	24,576	1
susan	0.670	0.675	1.01	253,952	253,952	1	253,952	1
Office								
stringsearch	0.024	0.024	1.00	16,384	16,384	1	16,384	1
Security								
sha	0.145	0.138	0.95	24,576	24,756	1	24,576	1
blowfish	0.713	0.722	1.01	24,576	24,756	1	24,576	1
rijndael	0.340	0.366	1.07	24,576	24,576	1	24,576	1
Network								
dijkstra	0.340	0.349	1.02	32,768	32,768	1	32,768	1
Telecomm								
CRC 32	1.463	1.53	1.04	16,384	16,384	1	16,384	1
adpcm codes	1.255	1.252	1.00	0	0	—	0	—
FFT	0.495	0.478	0.96	540,672	540,672	1	540,672	1
gsm	1.979	1.959	0.98	24,576	24,576	1	24,576	1
Multimedia								
g721	0.354	0.355	1.00	24,576	24,576	1	24,576	1
mpeg(decode)	0.331	0.320	0.97	385,024	401,408	1.04	401,408	1
epic	0.126	0.128	1.01	671,744	681,616	1.01	779,920	1.14
rasta	0.124	0.125	1.01	147,456	212,992	1.44	212,992	1

Exec. ratio is the ratio of execution time after pool allocation to the original time (A ratio of 2 means the program runs twice as long as the original).
Mem. ratio 1 is the ratio of the memory usage of program after pool allocation to that of the original program.
Mem. ratio 2 is the ratio of the memory usage of pool allocated program with our safety restriction to that of just the poolallocated program.

`epic`, and 19 of 80 in `rasta`.[5] In fact, all the case 3 pools in `mpeg2decode`, `rasta`, and `epic` 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 in these three benchmarks can be converted to case 1 or 2 with more sophisticated compiler analyses where the `pooldestroy` on a pool is moved as close to the last `poolfree` on the pool as possible without compromising safety [Lattner and Adve 2005].

Another interesting use of dynamic memory is seen in `dijkstra`, where a linked list is live throughout the program and the program repeatedly allocates and deallocates memory. In a language with explicit regions such as

[5]The specific number of pools and numbers of cases 1, 2 and 3 pools depend on the precision of DSA and pool allocation. We have made several improvements in DSA and pool allocation since our initial experiments [Dhurjati et al. 2003], leading to larger numbers of total pools and larger numbers of cases 2 and 3 pools in `epic`, `rasta`, and `mpeg2decode`. Nevertheless, the overall results are qualitatively similar, and our new measurements of memory consumption show that the impact on peak memory consumption of these codes is negligible.

Cyclone [Grossman et al. 2002] or RT-Java, this list would have to go on a garbage-collected heap or incur a potentially large memory increase. In our technique, the case 2 pool allows reuse of memory within the pool. 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.

6.3.1 *Evaluation of Run-time Costs of Heap Safety.* In Table III we present our results on evaluating the run-time costs of the heap safety approach. Since we do not insert any run-time array bounds checks in this work, the programs rejected by our array bounds checker are actually unsafe. We include their execution times only to show the effect of the pool allocation transformation on performance. First, we compared the execution times of these applications after the pool allocation transformation used for heap safety to the original execution time. Most of the execution times after pool allocation are within 2% of the original execution time, and only one program shows an increase of 7%. In some cases, we can even see that the pool allocation transformation improves the execution time. These results show that our heap safety mechanism only results in a marginal increase (if any) in execution time.

Next we measured the maximum memory usage of these programs. To identify the causes for increase or decrease in memory consumption, we measured the usage in three versions: original program, pool-allocated program, and pool-allocated program along with our safety restriction that memory cannot be released to the system until pooldestroy. Our other analyses (stack safety, array safety, and so on) do not change the memory consumption of the program. To better understand the numbers, we first give a brief description of our pool allocation run-time library. The library internally manages memory using malloc and free. To amortize the allocation costs over various allocation requests, it mallocs memory in multiples of pages of size 1K bytes. It releases memory to the system if it has more than a threshold number of free pages.

The column "Mem ratio 1" in Table III shows the memory increase due to pool allocation when compared to the original program. The increase is insignificant in most programs except rasta, where it is 44%. We found that rasta has many global pools which allocated a total memory of 8 bytes, while we were reserving a page of 1024 bytes for each such pool in our run-time library. In general, the total memory usage for these embedded programs is not high and any wastage (such as in `rasta`) appears large in percentage terms but its impact is minor in practice. If we decrease the page size to 512 bytes, we found that memory increase reduced to 28%, showing that a well-tuned pool allocation run-time library that dynamically increases page sizes depending on the allocation requests can do much better than our simple untuned version.

We then measured the additional increase in memory usage due to our safety restriction. As mem ratio 2 illustrates, only `epic`, which has a case 3 pool, shows an increase (of 14%) due to the additional safety restriction. Other programs with case 3 pools, `rasta` and `mpeg2decode`, do not show any increase.

Overall, our results indicate that case 3 pools occur infrequently even in complex embedded programs and typically never occur at all in simpler programs.

When it does occur, the increase in maximum memory usage seems acceptable. This is strong empirical evidence that our technique is powerful enough to enforce heap safety statically in a broad range of embedded programs.

6.4 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 stack safety column of Table II shows that only one program (`rijndael`) failed this check. This proved to be a false positive that occurred because data structure analysis is flow insensitive. 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.

6.5 Effectiveness of Array Access Checks

Our array bounds checker passed all the three classes of control programs, 8 of the 13 benchmarks from MiBench, and none from MediaBench, after the few changes described earlier. Interestingly, our tests detected four potential array bound violations in the MiBench suite and two in MediaBench: one each in `dijkstra` (both the large and small versions), `epic`, and `blowfish` and two violations in `g721`. Two of the errors, in `dijkstra` and `blowfish`, 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 stack corruption. The error in `epic` was an incomplete check before an array access.

The array bounds checking algorithm failed to prove safety for nine of the programs. Two of these programs used non-affine bit operations on the index variables. Five other programs use indirect indexing for arrays, for example, 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 programs 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 enforcing memory safety with no run-time software checks for a broad class of embedded applications.

7. RELATED WORK

Existing techniques for achieving memory safety can be differentiated based on the language properties that they enforce. On the one end of the spectrum lies software fault isolation (SFI) [Wahbe et al. 1993], which ignores all language-level information and enforces memory safety by *sand-boxing* every memory access/jump at run-time. SFI does not detect semantic errors such as array bounds errors, references to uninitialized values, or accesses to locations in dead stack

frames. Furthermore, SFI introduces significant and sometimes high run-time overhead, ranging from 25% to 59% when checking only write references and typically over 100% when checking both reads and writes. On the other end, languages like Java [Gosling et al. 2000], Cyclone [Jim et al. 2002], and others enforce strong type safety, which trivially ensures memory safety. Type safety helps in better reasoning/understanding of programs and early bug detection, but enforcing type safety typically involves placing some run-time checks on individual memory operations like array bounds checks, null pointer checks, and relying on garbage collection or programmer annotations (for region-based language extensions together with GC) for ensuring safe dynamic memory usage. Our approach lies closer to type-safe languages in the sense that we enforce type safety for most language features (which does not require run-time checks), but in order to eliminate the need for garbage collection, we permit dereferences of dangling pointers as long as they point to the correct type. This violates the technical definition of strong type safety.

A number of other approaches have been proposed to eliminate garbage collection and specific types of run-time overheads for type safe (and some non-type-safe) languages, and in the following we compare our approach with those.

The real-time specification for Java (RT Java) [Bollella and Gosling 2000] 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. Run-time 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 run-time 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 run-time 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 [Bollella and Gosling 2000]. Finally, RT Java also inherits the other run-time checking needs of standard Java such as for arrays, null pointer checks, and type coercions.

Real-time garbage collection techniques (e.g., see [Bacon et al. 2003], 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 a recent work [Bacon et al. 2003].

As an alternative to garbage collection, several recent languages (e.g., RT Java [Bollella and Gosling 2000], Cyclone [Grossman et al. 2002; Jim et al. 2002], and others [Boyapati et al. 2003; Gay and Aiken 1998]) have adopted mechanisms for region-based memory management. These languages disallow direct deallocation of items within a region in order to ensure program safety. 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 nonnested 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 [Grossman et al. 2002; Tofte and Birkedal 1998] can eliminate or mitigate the first but not the second, and has only been successful for type-safe languages without explicit deallocation. (This algorithm differs from automatic pool allocation in two important ways: it does not track object references across destructive heap updates, potentially allowing objects to be leaked, and it segregates objects primarily by lifetime, and does not internally segregate objects by data structure or points-to relationships.)

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 analyses. There are two potential disadvantages in our work, however. We do not prevent certain kinds of errors such as dangling pointer references. 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 [Lattner and Adve 2004]).

Boyapati et al. [2003] present a static type system combining ownership types with region types, to eliminate the run-time 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 "subregions" of a region for sharing region data safely across threads, using reference counts to reclaim the data. We do not support multithreaded applications so far.

Linear types and alias types [Crary et al. 1999; DeLine and Fahndrich 2001; Fahndrich and DeLine 2002; Walker and Morrisett 2001] have been used to prove memory safety statically 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 [DeLine and Fahndrich 2001], 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 check statically key correctness requirements of system calls and trusted libraries.

Many other systems including CCured [Condit et al. 2003; Necula et al. 2002] and SafeC [Austin et al. 1994; Jones and Kelly 1997 and the references therein], have tried the approach of adding meta data and inserting run-time *software* checks to make C programs safe. Most of these systems, except CCured, report prohibitively high overhead (up to 500%) to be of any use for production level software. Here, we compare our approach to two such systems, SafeC and CCured. SafeC uses a fat pointer representation to store spatial and temporal information for all pointers in a program and uses run-time software checks to detect memory errors. They report execution overheads up to 540% and space overheads up to 100%. These overheads are simply unacceptable in the context of embedded systems. In contrast, we do not need to insert any run-time

software checks except for array bound violations. CCured extends the C type system to infer statically verifiable safe pointers and adds run-time checks for other pointers. Our type safety rules, by not allowing casts between pointers, essentially restrict the C language to these safe pointers of CCured. Within this restricted language, our approach differs from CCured in a few fundamental ways. First, CCured relies on garbage collection for ensuring safe deallocation of heap memory, while we eliminate garbage collection altogether and provide a memory safety guarantee in the presence of dangling pointers (with no run-time overhead). Note that CCured does not detect dangling dereferences either: replacing manual memory deallocation by garbage collection in a program with a dangling pointer dereference error will only silently convert the error into a logical error. Also, unlike CCured we use static analysis to eliminate run-time checks for array bounds in some cases, we detect and avoid certain potential errors for stack references, and on some platforms we do not use any run-time software checks for uninitialized pointers.

A valuable strategy for compiler-based secure and reliable systems is proof-carrying code (PCC) [Necula 1997]. 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 which aspects of a program require static analysis and which require run-time 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 Bodik et al. [2000], Wagner et al. [2000], Gupta [1993]), 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. We use a constraint generation technique similar to the ABCD algorithm [Bodik et al. 2000] within a procedure, but go beyond them by developing an interprocedural constraint propagation algorithm. The work of Gupta [1993] tries to reduce the run-time overhead of bounds checks by eliminating partially redundant checks. Since we have focused on complete elimination of bounds checks, those techniques do not directly apply to our work. Wagner et al. [2000] have developed a tool for detection of buffer overrun vulnerabilities in general C programs. 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. CSSV [Dor et al. 2003] is another static bounds checking tool, which requires manual annotations to prove safety of array references. We do not use any programmer annotations but rely on our language restrictions and the interprocedural constraint propagation algorithm to prove array safety. Cousot and Halbwachs [1978] in their seminal work present an abstract interpretation technique to compute affine relationship among variables of a program at every program point within a procedure using an iterative fix point analysis for

convergence. While our symbolic analysis can also be considered as one case of abstract interpretation, the use of def-use chains of static single assignment (SSA) form along with traditional induction variable analysis helps us to avoid iteration for loops and ignore unrelated constraints. Use of SSA form for efficient intraprocedural symbolic analysis is not new [Bodik et al. 2000; Tu and Padua 1995]. Our main contribution is in extending the symbolic analysis to the interprocedural case without losing context sensitivity.

8. CONCLUSIONS AND FUTURE WORK

The broad approach of the work presented here has been to identify minimal semantic restrictions on imperative programs and to develop new compiler techniques that together enable memory safety to be enforced without using garbage collection or run-time checks (on systems with hardware memory protection). To our knowledge *no other programming language or compiler system achieves this goal for any nontrivial 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.

More specifically, we identified four challenges to ensuring memory safety—detecting uninitialized pointer dereferences, stack safety, array bounds checking, and heap safety. We addressed each of these problems using minimal semantic restrictions on programs and some novel compiler techniques. One of the key new results in this work is to show how an automatic pool allocation transformation allows us to ensure that dereferencing dangling pointers to freed memory does not cause any violations of memory safety *without programmer annotations, run-time checks, or garbage collection*. An additional 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 20 programs we studied without causing any significant increase in memory consumption. We also developed a technique for eliminating null pointer software checks by converting them into hardware checks, a static check for stack safety, as well as a new interprocedural array bounds checking algorithm that exploits our semantic restrictions. Our checks for null pointers dereferences and stack safety are also effective for nearly all these programs, but our current analysis for checking array references can do complete static checking for only half the benchmarks we studied.

Overall, as shown in some of our results, the programs certified as safe by our compiler can execute as fast as 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). Furthermore, although our presentation and examples have focused on C, the semantic restrictions are defined on a low-level language-independent type system and instruction set and are implemented in a language-independent, link-time compiler framework; and therefore they should be applicable to other similar imperative languages.

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 run-time environment with mechanisms to enforce correct usage of system calls and run-time libraries. Finally, we aim to extend our techniques so that they can be used for arbitrary programs, including non-type-safe code that is relatively common in low-level system software, by introducing some additional run-time checks but still without requiring automatic memory management (i.e., permitting explicit memory deallocation).

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman, Reading, MA.

AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. 1994. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Language Design and Implementation*. 290–301.

BACHMANN, O., WANG, P. S., AND ZIMA, E. V. 1994. Chains of recurrences—A method to expedite the evaluation of closed-form functions. In *International Symposium on Symbolic and Algebraic Computation*. 242–249.

BACON, D., CHENG, P., AND RAJAN, V. 2003. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of 30th ACM Symposium on Principles of Programming Languages (POPL03)*. 285–298.

BIRCH, J. 2002. Using the chains of recurrences algebra for data dependence testing and induction variable substitution. M.S. thesis, Computer Science Dept., Florida State University, Tallahassee, FL.

BODIK, R., GUPTA, R., AND SARKAR, V. 2000. ABCD: Eliminating array bounds checks on demand. In *SIGPLAN Conference on Programming Language Design and Implementation*. 321–333.

BOLLELLA, G. AND GOSLING, J. 2000. The real-time specification for Java. *Computer 33*, 6, 47–54.

BOYAPATI, C., SALCIANU, A., BEEBEE, W., AND RINARD, M. 2003. Ownership types for safe region-based memory management in real-time Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, 324–337.

CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G. 1992. Modula3 language definition. *ACM Sigplan Not. 27*, 8 (Aug.).

CHIN, W.-N., CRACIUN, F., QIN, S., AND RINARD, M. 2004. Region inference for an object-oriented language. *SIGPLAN Not. 39*, 6, 243–254.

CONDIT, J., HARREN, M., MCPEAK, S., NECULA, G. C., AND WEIMER, W. 2003. CCured in the real world. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, 232–244.

COUSOT, P. AND HALBWACHS, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Tucson, AZ. ACM Press, New York, 84–97.

CRARY, K., WALKER, D., AND MORRISETT, G. 1999. 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, TX. ACM, New York, 262–275.

DELINE, R. AND FAHNDRICH, M. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*. Snowbird, UT. 59–69.

DHURJATI, D., KOWSHIK, S., ADVE, V., AND LATTNER, C. 2003. Memory safety without runtime checks or garbage collection. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*. San Diego, CA, 69–80.

DOR, N., RODEH, M., AND SAGIV, M. 2003. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, 155–167.

FAHNDRICH, M. AND DELINE, R. 2002. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*. 13–24.

GAY, D. AND AIKEN, A. 1998. Memory management with explicit regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada. 313–323.

GORDON, A. D. AND SYME, D. 2001. Typing a multi-language intermediate code. *ACM SIGPLAN Notices. 36*, 3, 248–260.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *The Java Language Specification*. Sun Microsystems.

GROSSMAN, D., MORRISETT, G., JIM, T., HICKS, M., WANG, Y., AND CHENEY, J. 2002. Region-based memory management in Cyclone. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*. 282–293.

GUPTA, R. 1993. Optimizing array bound checks using flow analysis. *ACM Lett. Prog. Lang. Syst. 2*, 1-4 (Mar.–Dec.), 135–150.

GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, 1–12.

HEINE, D. L. AND LAM, M. S. 2003. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. New York, 168–181.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. 2000. System architecture directions for network sensors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 93–104.

JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. 2002. Cyclone: A safe dialect of C. In *Proceedings of USENIX Annual Technical Conference*, 275–288.

JONES, R. W. M. AND KELLY, P. H. J. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*. 13–26.

KELLY, W., MASLOV, V., PUGH, W., ROSSER, E., SHPEISMAN, T., AND WONNACOTT, D. 1996. *The Omega Library Interface Guide*. Tech. Rep., Computer Science Dept., U. Maryland, College Park. Apr.

LATTNER, C. 2002. LLVM: An infrastructure for multi-stage optimization. M.S. thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. See `http://llvm.cs.uiuc.edu`.

LATTNER, C. AND ADVE, V. 2005. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, IL.

LATTNER, C. AND ADVE, V. 2003. *Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis*. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign.

LATTNER, C. AND ADVE, V. 2004. LLVM: A complilation framework for lifelong program analysis and transformation. In *Proceedings of the Second International Conference on Code Generation and Optimization*. Palo Alto, CA. 75–86.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*. 330–335.

LEVIS, P. AND CULLER, D. 2002. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA. 85–95.

NECULA, G. C. 1997. Proof-carrying code. In *Proceedings of of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, Paris. 106–119.

NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of 29th ACM Symposium on Principles of Programming Languages (POPL '02)*, London. 128–139.

OAKS, S. 2001. *Java Security*, 2nd ed. O'Reilly. ISBN 0-596-00157-6.

PATIL, H. AND FISCHER, C. 1995. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software–Practice and Experience 27*, 1, 87–110.

PUGH, W. 1992. A practical algorithm for exact array dependence analysis. *Commun. ACM 35*, 8 (Aug.), 102–114.

SHA, L. 1998. Dependable system upgrades. In *Proceedings of IEEE Real-Time System Symposium*, 440.

SHA, L. 2001. Using simplicity to control complexity. *IEEE Software*, 20–28.

TOFTE, M. AND BIRKEDAL, L. 1998. A region inference algorithm. *ACM Trans. Prog. Lang. Sys. 20*, 1, 724–767.

TOFTE, M. AND TALPIN, J.-P. 1997. Region-based memory management. *Inform. Comput. 132*,2, 109–176.

TU, P. AND PADUA, D. A. 1995. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *International Conference on Supercomputing*. 414–423.

WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego, CA. 3–17.

WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. 1993. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review 27*, 5 (December), 203–216.

WALKER, D. AND MORRISETT, G. 2001. Alias types for recursive data structures. *Lecture Notes in Computer Science* Vol. 2071, 177+.

Received November 2003; revised April 2004, August 2004; accepted August 2004