Ensuring Code Safety Without Runtime Checks for Real-Time Control Systems *

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

Abstract. This paper considers the problem of providing safe programming support and enabling secure online software upgrades for control software in real-time control systems. In such systems, offline techniques for ensuring code safety are greatly preferable to online techniques. We propose a language called Control-C that is essentially a subset of C, but with key restrictions designed to ensure that memory safety of code can be verified *entirely* by static checking, under certain system assumptions. The language permits pointer-based data structures, restricted dynamic memory allocation, and restricted array operations, without requiring any runtime checks on memory operations and without garbage collection. The language restrictions have been chosen based on an understanding of both compiler technology and the needs of real-time control systems. The paper describes the language design and a compiler implementation for Control-C. We use control codes from three different experimental control systems to evaluate the suitability of the language for these codes, the effort required to port them to Control-C, and the effectiveness of the compiler in detecting a wide range of potential security violations for one of the systems.

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

Keywords

real-time, control, compiler, programming language, static analysis, security

*This work is supported in part by an NSF CAREER award, grant number EIA-0093426, and by the University of Illinois, and in part by the Office of Naval Research N0004-02-0102, and in part by DARPA's NEST program.

CASES 2002, October 8–11, 2002, Grenoble, France.

1. INTRODUCTION

This paper considers the problem of providing safe programming support and enabling secure online software upgrades for control software in real-time control systems. The main goal of this work is to design a language based on C but with key restrictions that ensure that the safety of code written in the language can be verified entirely by static (i.e., compile-time) checking, *without requiring any checks during program execution, and without garbage collection*. This goal of 100% static checking distinguishes our work from a wide range of existing safe languages for general-purpose or real-time systems. The language restrictions have been carefully chosen with an understanding of both compiler technology and the needs of real-time control systems, in order to achieve a language design that is not too restrictive for the class of target applications, yet not too powerful to allow static checking of memory safety. Below, we first motivate the need for program safety in real-time control systems, discuss why existing approaches are insufficient, and then summarize the results of this work.

Many physical systems in manufacturing, automotive transport, air transport, and others are increasingly being put under the control of sophisticated computerized control systems. An increasingly important need in embedded control systems is the ability to perform online upgrades, because embedded systems have long life cycles and because the downtime of a large distributed control system is often very expensive and sometimes impractical. This requires the ability to replace one or more components of the control software of a large real-time control system without shutting down the system. Furthermore, because of the critical nature of such systems, and because of the complexity of control software, it is important that the systems must remain stable and operational even if the software "upgrades" contain inadvertent bugs or deliberate attacks. In other words, new software components introduced online must be treated as untrusted.

Other research projects are working on the control-theoretic and real-time scheduling challenges in dynamically upgrading a control system in the field. The Simplex architecture [21, 22] is a real-time fault-tolerant system designed to support dependable system upgrade. It allows for online replacement of control software without shutting down system operations and it tolerates arbitrary errors in application-level *timing* and *control* logic. Nevertheless, this architecture is vulnerable to *software* bugs or attacks such as due to address space violations or illegal system calls hidden in the code. (We refer to these collectively as memory errors.) As explained in detail in Section 2.1, in order to ensure system safety without major performance loss, the system must protect against such errors within individual control threads that could potentially compromise the operation of the other modules within the same address space.

When ensuring against memory errors in real-time control systems, offline techniques are greatly preferable to online techniques for two important reasons. First, it is important not to introduce significant or unpredictable runtime overhead in control codes since sophisticated control systems often have to operate under extremely tight real-time scheduling constraints. Second, and perhaps even more important, it is usually far more cost-effective (and perhaps essential) to detect software errors during development than during actual operation.

There has been extensive research on language, compiler, and operating system techniques for ensuring program safety under different types of constraints. Unfortunately, it appears that current language or system designs require significant overheads in terms of runtime checks and garbage collection for ensuring code safety. Safe languages like Java [8], Modula-3, ML, Safe-C [2] and CCured [19]. use various runtime checks such as for array bounds checking, null pointer references, and type coercions, and also rely on runtime garbage collection to help ensure the safety of pointer dereferences. RT-Java [4] provides a safe real-time language using incremental garbage collection algorithms under real-time schedulers but these do not reduce the runtime overheads of garbage collection. In fact, RT Java includes three additional flavors of dynamic memory that are not garbage collected, but which require extensive runtime checks for object references between different memory flavors. As discussed in Section 6, several research languages have been proposed that use new annotation mechanisms or language restrictions to reduce the need for runtime checking, but none appear to come near our goal of entirely eliminating runtime checks on memory operations [12, 9, 6, 17, 18]. It is important to note that the overheads of runtime safety checking are indeed significant: languages like SafeC, CCured, Cyclone and Vault have reported slowdowns ranging from 20% up to 2x or 3x for different applications [2, 19, 9, 6].

The aim of this work is to design a programming language based on C that is restrictive enough to allow complete static checking of the memory safety of a program, while still being flexible enough to support sophisticated real-time control applications and potentially other similar embedded software. (We rely on some system support to trap accesses to a range of reserved addresses, in order to eliminate all runtime checks; otherwise, some NULL pointer runtime checks are required at runtime.) In addition to some basic restrictions for type safety, we introduce two novel restrictions to ensure safety properties that are otherwise very difficult to verify statically: dynamic memory allocation, and array accesses. We restrict dynamic memory allocation to use a region-based allocation method [23, 7], but restrict it to *a single dynamic region at a time* in order to allow static checking without requiring significant new annotation mechanisms in the language, such as those used in some recent systems [6, 9]. We restrict array usage in a program so that array index expressions, the loop bounds of relevant enclosing loops, and corresponding array extents are restricted functions of each other. The restrictions are a compromise between expressiveness and the capabilities of current integer programming techniques used in static analysis [20, 13].

A key feature we chose to omit is annotations on pointers, function interfaces and function calls, such as the annotations used in other recent work [6, 9]. Our motivation is to minimize the need for new language mechanisms that can require significant changes to existing application level code. In fact, except for three new intrinsics for region-based memory allocation (replacing `malloc` and `free`), Control-C is a strict subset of C. Instead of annotations, the language design relies on aggressive compiler technology to perform the necessary static safety checks. This paper gives a brief overview of the compiler techniques we have used to implement the language, and discusses tradeoffs in the language design motivated by these techniques. The compiler includes two new interprocedural algorithms for checking safety of array accesses and of pointer references (including accesses to dynamically allocated region memory). The details and evaluation of these algorithms are beyond the scope of this paper. *Together, the language restrictions described here enable complete static safety checking for C programs without requiring new annotation mechanisms, runtime checks, or garbage collection, and assuming only the system support noted above.*

Although the language proposed here is based on C, our compiler implementation operates on a low-level typed virtual instruction set (LLVM) to enforce the restrictions above [15]. This means that type-safe code in other languages (e.g., C++ or Fortran) meeting the restrictions described above on type safety, pointers, arrays, and dynamic allocation could also be used directly.

The next section provides some background on the Simplex real time system architecture, and on the LLVM compilation framework. The following two sections describe the language design (Section 3) and the compiler implementation (Section 4). Section 5 evaluates the the expressiveness of the language supporting legal control codes, and the effectiveness of the language and compiler in detecting a variety of different safety violations. Section 6 compares our work with previous approaches, and Section 7 concludes with a summary and directions for further work.

2. BACKGROUND

2.1 The Simplex Real-Time System Architecture

The work described in this paper is a collaboration between the compiler group and the real-time systems group that developed the Simplex architecture. The *Simplex* architecture [21, 22] is a dynamic real-time fault-tolerant middleware system. It supports online replacement of control software without shutting down system operations, and tolerates arbitrary timing errors and application-level logical errors in the upgraded control software. Also, using limited address protection, it tolerates run-time software faults trapped by the operating system such as segmentation faults and divide-by-zero errors.

A testbed known as Telelab demonstrates the Simplex architecture (see `http://www-rtsl.cs.uiuc.edu/drii`). Telelab uses a computer-controlled inverted pendulum (IP) as a model of a control system, and allows users anywhere on the Internet to experiment with dynamic upgrades of the control code. Telelab's embedded Simplex environment can safely run untrusted new control software sent to the system, irrespective of where it comes from and what algorithm it uses. Readers may download the interface at `http://www-rtsl.cs.uiuc.edu/drii/download`. The language and compiler described in this paper are being incorporated into the online Telelab testbed to make them available for public experimentation and use.

Although Simplex tolerates some memory errors, recovering from any error (timing, logical, or memory) incurs a severe performance penalty. Furthermore, because the process-level protection is limited, Simplex is still vulnerable to memory errors or attacks such as illegal writes to the address space of the backup controller or a `kill` system call that brings down user-level processes. Also, many embedded platforms offer no address space protection.

Our goal for a dynamically upgradable real-time platform like Simplex is to use a light-weight thread-based run-time architecture with separate threads for each control module and thread-level component replacement. A compiler should enforce thread-level fault-isolation, and in particular must ensure that illegal memory operations or system calls in one thread do not compromise the correct functioning of other threads in the system.

2.2 The LLVM Virtual Instruction Set and Compilation Framework

Our work has been implemented within the LLVM compilation system. LLVM — Low Level Virtual Machine — is a typed instruction set and compilation framework designed to enable high-level optimizations at *link-time*, *runtime*, and *offline* [15]. The key idea in LLVM is to use a rich virtual instruction set (instead of raw machine code) as the object code representation manipulated by link-time and post-link optimizers and code generators. The instruction set uses low-level RISC-like operations, but provides rich type and dataflow information about the operands. Together, these enable high-level (but language-independent) program analyses and transformations on object code from arbitrary high-level languages.

The LLVM instruction set uses an infinite set of typed virtual registers that are in Static Single Assignment (SSA) form [5]. `Load` and `store` operations via typed pointers are used to transfer values between memory locations and registers. The types in LLVM include primitive integer and floating point types and basic derived types (pointers, arrays, and user-defined structures). All register and memory operations are strictly typed, and the only mechanism to violate type constraints (e.g., in order to add an integer and a floating point value) is by using a special `cast` instruction.

The LLVM compiler infrastructure includes a GCC-based front-end that compiles C programs to LLVM object code (C++ will be supported in the near future), an interprocedural link-time optimizer, and two back-ends: one for the SPARC v9 architecture, and one that generates portable C code. The Control-C compiler described in this paper is implemented within the LLVM link-time compilation system.

3. THE CONTROL-C LANGUAGE

The design choices in the Control-C language are essentially trade-offs between three different concerns;(a) the ability to guarantee memory safety through static analysis, (b) maximizing the expressiveness of the language (for real-time control applications in particular) and (c) the ease of porting existing applications.

The choices we make are motivated by the general code structure of embedded real-time (especially, control) applications. Such codes generally consist of one of more tasks that have to execute within a given time bound prescribed by a real-time scheduler, and continuing *ad infinitum*. The tasks are typically computationally intensive. The data structures used in the task are usually dense arrays, which may be allocated statically or dynamically. Dynamically allocated data would usually be allocated before the timed loop is entered, or would allocate memory in the beginning of each loop iteration and free it at the end of the iteration. Some tasks may use pointer-based data structures, but these are usually relatively simple. Most of the existing embedded codes are written in C.

Based on these observations, we designed Control-C essentially as a subset of the C language (except for three new intrinsic functions, replacing `malloc` and `free`). Control-C imposes several language restrictions in order to ensure that static checking of program safety is possible. In particular, static safety checking at compile time must accomplish three goals (In the following, the terms "program" or "application" refer to the collection of source code provided to the compiler. Typically this will represent all the source code for a single controller or a single control module in a hybrid controller):

- Ensuring that there are no memory references beyond the locations explicitly allocated (statically or dynamically) by the program.
- Ensuring that all control flow in the code is legal and safe, e.g., there are no attempts to branch to addresses in data areas.
- Disallowing the program from invoking any untrusted functions not included in the target source code.

Any violation of these principles can allow malicious code to compromise a system and gain illegitimate access. Illustrations of such malicious code can be found in section 5.2.

In designing the language, we made a few key assumptions about the underlying embedded system:

(S1) Certain errors (listed below) are acceptable at runtime. If a runtime error in the control application is correctly detected, the underlying system has the ability to recover from the error and restore control to a backup controller. We refer to this as a *safe runtime error*.

(S2) Stack overflow (e.g., due to infinite recursion) and heap overflow due to dynamic allocation generate safe runtime errors.

(S3) The system reserves a range of addresses for which any access causes a safe runtime error. (If this is not available, some null pointer checks must be inserted in the code, as described in Section 4.1.)

(S4) A set of trusted functions is known to the compiler and these are assumed to be safe to invoke. We require the source code to be provided for any untrusted function invoked by the target code.

Checking for the safe runtime errors above should typically require negligible overhead. Simplex is already designed to tolerate detected errors – when a runtime error (of any kind) is detected by the runtime system, Simplex kills the offending controller and restores control to a trusted backup controller. With these assumptions we are able to focus on static checking of memory accesses that are usually much more expensive to check at runtime.

The restrictions in Control-C fall in three categories: basic restrictions for type safety, array operations, and dynamic memory allocation. These are described in the next three subsections.

3.1 Basic Issues

The first set of restrictions in Control-C (relative to C) focus on type safety, accesses to uninitialized pointers and some other basic issues.

(T1) The Control-C language requires strong typing of all functions, variables, assignments, and expressions, using the same types as in C.

(T2) The language disallows casts to or from any pointer type. Casts between other types (e.g., integers, floating point numbers, and characters) are allowed.

(T3) A union can only contain types that can be cast to each other; consequently a union cannot contain a pointer type.

(T4) The language requires that there are no uses of uninitialized local pointer variables within a procedure. In particular, a pointer variable must be assigned a value *before it is used or its address is taken*.

(T5) If the application chooses to exploit assumption (S3) above to avoid runtime NULL pointer checks, then any individual data object (scalar, structure, or array, allocated statically or dynamically) should be no larger than the size of the reserved address range.

(T6) Pointer arithmetic is disallowed in Control-C.

Explicit array declarations are just as in C, i.e., they need to specify the size of each dimension, except for the first dimension of an array formal parameter.

To motivate T4 above, consider the following code snippet, which is disallowed in our language:

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

Here, the address of uninitialized pointer `p` is assigned to `pp`. Now dereferencing `p` via `**pp` is potentially unsafe and this violation would be difficult to detect. The language therefore disallows taking the address of an uninitialized pointer.

Rule T5 is needed to avoid the need for checking uninitialized pointer values within global or dynamically allocated aggregate objects, as explained in Section 4.1. If the programmer does not want to restrict the size of the objects used in the program, this restriction can be ignored. However, this would mean that runtime checks for NULL pointers cannot be avoided. This choice offered to the programmer (specified through a compiler option) is further discussed in Section 4.1.

Language rule T6 above disallows pointer arithmetic. The C language allows pointer arithmetic only for array traversal, and any other form of pointer arithmetic is undefined. Array traversals must be done using explicit array indexing in Control-C. We make this choice because interprocedural analysis for array bounds checks may become significantly more difficult if explicit pointer arithmetic is allowed. Note that traversing a string now requires using `strlen` followed by explicit array index operations. We do not currently permit other string library operations and we aim to provide a safe string library in the future.

3.2 Restrictions on Array Operations

In general, array operations are one of the most expensive to check for memory safety at runtime. Our approach is to impose 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.

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, 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 [13] provides an efficient implementation that has been widely used for solving such integer programming problems in compiler research.) Exploiting static analysis based on Presburger arithmetic requires that our language only allows linear expressions with constant (known) 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).

Array operations in Control-C must obey the following rules. 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 has to be a positive expression.

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

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

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

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

(A4) If an array is accessed outside of a loop then

(a) the index expression of the array has to be provably an affine expression of the length of the array.

A1 by itself guarantees safe array accesses, but the compiler can check that a program satisfies A1 only if the additional language rules A2—A4 are obeyed.

Note that the length of an array can be any non-negative 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.

Rule A3(c) 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 $K..N + K - 1$ 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(c) 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.

To illustrate the rules further, consider the code shown in fig. 1:

This code fragment is a valid Control-C code because

1. `n=20` can be proven to be an affine function of the size of `B` (A4);
2. `m*4` is clearly an affine function of `m` (A3(b)); and
3. the bounds of the loop (`0` and `B[n]`) can be proven to be affine functions of the size of `A` (A3(a)).

```
... /*** caller function ***/
if (( k > 0) && (k < 5)) {
  B = (int *)RMalloc(k * 30);
  C = initialize(B, 20);
}

int * initialize(int *B, int n) {
  int *A,m;
  if (B[n] > 0) {
    A = (int *) RMalloc(5 * B[n] + 10);
    for (m=0; m < B[n]; ++m) A[m*4] = m ;
    return A;
  } else return null;
}
```

Figure 1: Array Usage Example

Once the above three conditions are satisfied, the compiler proves A1 (i.e., the code is safe) for all array accesses. Note that for proving safety of B[n], the compiler has to correctly include the constraint on k ($k < 5$) and only then it can verify that n (= 20) is less than the size of B.

3.3 Regions and Dynamic Memory Allocation

Control-C supports region-based memory allocation [23] where only a single region may be active in the program at any given time. An arbitrary amount of memory may be allocated out of a region (with some system-dependent limit), using a `malloc`-like interface. There is no `free` operation for individual memory objects; the entire region must be freed at once.

The language provides three intrinsic functions (replacing `malloc` and `free`) for region-based memory management: `RInit`, `RFree`, and `RMalloc`. The region is made active by calling `RInit()` and freed (made inactive) using `RFree()`. `RMalloc` has the same signature as `malloc`, and allocates memory within the active region.

The constraints imposed by Control-C to allow safe dynamic memory allocation are summarized below.

(R1) Only one region is active at any point in the program. Thus calls to `RInit` and `RFree` must alternate on every potential path in the program, starting with an `RInit`.

(R2) The region should be active at any call to RMalloc.

(R3) Because region memory must not be accessed after a region is freed, any pointer value that contains a region address must be provably dead (unused or unreachable) at a call to `RFree`. To verify this statically we have the following constraints

(a) Any local and global scalar pointer variable must be explicitly reinitialized following a call to `RFree`, before any potential uses of the variable. Note in particular that this includes variables that *never* point to the heap (see the discussion below).

(b) Structures or arrays containing pointers must be allocated dynamically, either via `RMalloc` (on the heap) or via `alloca` (on the stack). In particular, aggregate objects containing pointers cannot live in global or local variables (with the exception of initialized constants).

Rule R3(a) above ensures that the compiler can statically prove that all scalar pointer variables are dead at a call to `RFree`. Rule R3(b) is necessary because tracking the contents of aggregate objects (e.g., an array of structures containing pointer fields) is generally difficult. Note that all dynamically allocated memory (either from `RMalloc` or `alloca`) can be accessed only via other pointer variables, and therefore such accesses must originate from some scalar pointer variable. R3(b) with R3(a) ensures that any pointer values within aggregate objects become unusable at an `RFree`.

The language guarantees that programs following the above rules will be considered safe by any conforming Control-C compiler. In practice, however, these rules could introduce significant overhead into the program. For instance, global pointers normally initialized at the beginning of the program would need to be reinitialized if an `RFree` occurred in the control loop, even if these pointers never point into the heap region.

In practice, a reasonably powerful compiler can prove that many pointer values are never used either to point to region objects, or to store region pointers into other pointer locations. A Control-C compiler may allow such values to remain live at a call to `RFree`, i.e., to be used without reinitialization. Although such programs are not completely portable (i.e., may be rejected by a different Control-C compiler with weaker analysis capabilities), the performance benefits may justify the potential extra porting cost. We therefore define two weaker versions of the above rules:

(R3′(a)) Any local and global scalar pointer variable that may hold an address within a heap region must be explicitly reinitialized following a call to `RFree`, before any potential uses of the variable.

(R3′(b)) Pointer fields are permissible within global or local aggregate objects only if they provably never hold an address within a heap region, and their address is never taken. All other aggregate objects containing pointer fields must be allocated dynamically, either via `RMalloc`) or via `alloca`.

A program observing these rules may be considered safe by some compilers, and can avoid the additional runtime overheads of reinitializing such pointer values at every `RFree`.

Our approach eliminates the need for any explicit "region" annotations on pointers by the programmer. Other approaches like Cyclone [9] and Vault [6] need functional annotations which describe the areas of memory a pointer may point to, since regions commonly are live across procedures and dynamically allocated data structures are passed between functions. Using compiler analysis over annotations is enabled because we have only a single active region at any time. This has the advantage however, that it is relatively simpler to port C control application code to Control-C. We illustrate the language features with some examples below:

```
...
if (cond) {
   t = 0;
   RInit();
} else {
  t = 1;
  RInit();
}
p = RMalloc(4);
RFree();
```

Example 1

```
if (cond) {
   RInit()
   ...
} else ...

if (cond) ...
else {
  RInit()
  ...
}
RFree();
```

Example 2

```
int *p,*q,*t;
int **r;
int a=2;
RInit();
t=&a;
p = RMalloc(
  4*sizeof(int));
q = p;
r = &p;
RFree();
...
```

Example 3

The first example shown above is valid, since at any point in the program only a single region is active. The second example is invalid according to our language restrictions since there exists a potential path with consecutive `RInit` calls. In the third code snippet, after the `RFree` call, if the pointer variables, `p`, `q`, `r` and `t` are necessarily re-initialized before they are used again, the program is guaranteed to be safe. However, if the compiler can prove that `t` necessarily does not point to the heap, then the language allows `t` to be used again without needing any re-initialization.

4. COMPILER ANALYSIS

In this section, we discuss our implementation of the Control-C compiler, with brief overviews of the new algorithms involved. As discussed in section 2.2, we have implemented our compiler within the LLVM compiler infrastructure. The type system in LLVM, the SSA representation, and the fact that this is a link-time compiler, all help to simplify the compiler analyses. Since LLVM is source-language-independent, our analyses can have broader applicability beyond C programs, though the Control-C language has been presented as restrictions relative to C. In each of the subsections below, we describe our algorithms to implement our language checks.

4.1 Uninitialized Variables and Type Safety

Because LLVM operations are strictly typed and the `cast` instruction is the only way of violating type constraints, checking the basic type safety rules of Control-C is straightforward. Our compiler simply checks all explicit `cast` instructions to ensure that there are no illegal casts (in particular, any casts involving a pointer type). We also check all arithmetic operations to ensure that no arithmetic is performed on pointer types.

Our uninitialized pointer analysis is a relatively simple intraprocedural dataflow problem that considers only local scalar pointer variables within a procedure. (Note that interprocedural analysis is not required for identifying uninitialized variables.) The analysis checks if such a variable has been stored to on every path to an instruction that dereferences the pointer or takes its address. The language requires that such stores be explicit (i.e., not via an alias).

Non-scalar pointers (i.e., pointer fields within aggregate objects) are very difficult to analyze and need an alternative approach. All such memory locations are initialized specially, as follows. Assume the range of addresses $[A : A + N - 1]$ is reserved (by assumption S3 in Section 3), and a reference by user-level code to any address in this range is trapped by the operating system. (For example, the high end of the user address space is reserved for the kernel in standard Linux implementations, typically 1 GB out of 4 GB, i.e., A =0xc0000000 and $N = 2^{30}$.) Then by rule T5, all global and dynamic memory locations are initialized to A, and every individual data object is restricted to be at most N bytes in size. (This assumption should be reasonable for embedded codes and large N.). This ensures that any typed reference through an uninitialized pointer (e.g., `p->fld` or `p[i]`, where $p == A$) will be illegal and trapped by the operating system.

Many current embedded systems have no address protection and may also have small (4, 8, or 16-bit) address spaces, so the above strategy cannot be used. Instead, the programmer can choose to ignore the size restriction in language rule T5. In that case, all global and dynamic memory locations will be initialized to NULL and runtime NULL-pointer checks will be required for such pointer accesses. We expect, however, that dynamic upgrading of code will only occur in high-end embedded platforms with more capable processors and operating environments, where the previous assumptions would apply and no runtime checks would be needed.

4.2 Checking for safe array usage

Compiler checking for safe array usage requires 3 steps:

- Generating constraints from each procedure,
- interprocedural propagation of constraints, and
- verifying whether each array access is safe.

4.2.1 Generating the constraints

We generate a set of constraints for each array access in a program, including only those constraints which affect the array access. We use a flow-insensitive algorithm that exploits the SSA representation in LLVM. For an array access, A[i][j], we traverse def-use chains backwards to get constraints from the definitions of A, i and j respectively. These constraints are simply inequalities that can be inferred from the program statements. For most program statements, generating the constraints is straightforward. For e.g., from a simple statement like `i = (x + z) * 5`, we would generate an affine constraint $i = 5x + 5z$. No constraints are generated for any non-affine expression. Note that not generating a constraint for an SSA-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 x and z, stopping only if we encounter a non-affine expression, a formal argument, a return value from a call, or a statement whose constraints have already been computed and cached. We cache the final constraints on each statement we traverse so that they can be reused.

Control-flow statements pose a problem. Consider the SSA code snippet on the left below: Clearly the first array access `A[x]` is

SSA	e-SSA [3]

A = (int *) RMalloc(20);
if (x < 20) {
 	
    A[x] = 100;
} else {
 	
    A[x+1] = 200;
}
	A = (int *) RMalloc(20);
	if (x < 20) {
	    $x_1 = \pi(x)$;
	    A[x_1] = 100;
	} else {
	    $x_2 = \pi(x)$;
	    A[x_2+1] = 200;
	}

Figure 2: Constructuing e-SSA from SSA

safe, and the second access $A[x + 1]$ is unsafe. These facts result from the branch condition $\texttt{x} < 20$ and the size of array `A`. We need to encode the "control dependence" information that $\texttt{x} < 20$ in the then branch and $\texttt{x} >= 20$ in the else branch in our flow insensitive constraint system. We cannot union both constraints together as it would make the system inconsistent. We use a technique developed in the ABCD algorithm [3], in which they encode the control dependence by inserting new nodes denoted π nodes. As shown on the right in figure 2, π nodes essentially provide different names within each branch for each variable appearing in the conditional expression. This representation, called extended-SSA (e-SSA) form in the ABCD paper, allows us to generate two simultaneous constraints $\texttt{x}_1 < 20$ and $\texttt{x}_2 >= 20$. Thus, for the above code snippet, the first array reference generates the constraints $\{A_{\text{length}} = 20 \wedge x_1 = x \wedge x_1 < 20\}$ and the second reference generates: $\{A_{\text{length}} = 20 \wedge x_2 = x \wedge x_2 \leq 20\}$.

For a ϕ node, $x_3 = \phi(x_1, x_2)$ in the SSA form, we first check if it is an induction variable, using standard induction variable analysis. If the ϕ node is not an induction variable, then we simply add the OR constraint $((x_3 = x_1) \vee (x_3 = x_2))$. With an induction variable of a loop, simply adding this OR constraint would result in an inconsistent system, since the ϕ merges values from a back edge and a forward edge. Instead, we check the step function of the induction variable. If the step function is positive then we add the constraint $((x_3 >= x_1) \vee (x_3 <= x_2))$, where x_1 comes from a forward edge and x_2 comes from a backward edge. If the step function is negative, then we add the constraint $(x_3 <= x_1) \vee (x_3 >= x_2)$. (Note that an induction variable for a loop with an unknown step function cannot represent an affine constraint, and will simply be ignored.)

4.2.2 *Interprocedural Propagation*

Since arrays are often declared in one procedure and passed as parameters to other procedures, it is essential to propagate constraints for arrays from call-sites to the callees. Also, constraints for an array access could depend on the return values of some procedures that are invoked earlier in some execution path. Therefore, constraints on the return values in terms of the incoming arguments must be propagated from the callee to the call statement.

We have developed an efficient algorithm for interprocedural propagation, which we describe very briefly here. A brute force propagation of constraints along each path in the call graph would produce an exponential algorithm since there could be an exponential number of paths in the call graph. Instead, by merging the constraints on incoming arguments from all call sites for for each procedure, we are able to achieve an algorithm with worst-case complexity of O(n^3), where n is the number of variables in the program. This worst case appears very unlikely to occur in realistic codes. In practice, we have found that a simple heuristic like collecting all the constraints for each of the possible different arrays passed to the procedure, then merging and simplifying them, removes many redundant constraints and greatly increases efficiency.

The interprocedural algorithm consists of two passes on the call-graph. First, a bottom-up pass gets the constraints on return values in terms of procedure arguments. A top down pass then merges constraints on arguments coming in to that procedure from different call-sites and then tries to prove safety for all array accesses in that procedure.

4.2.3 *Checking for array bound violations*

Once we generate the constraints for the array accesses, we use the Omega integer set library to test each array index expression, $A[exp]$, for safety. This just translates to checking whether satisfiability of the constraint system along with the added constraint $exp >= A_{\text{length}}$. If the system is unsatisfiable, then $exp < A_{\text{length}}$ and hence the array access is safe.

4.3 Region Checks

Compiler checking for region-based dynamic memory allocation requires checking that:

1. at most one region is active at a given time, and that a region is active at a call to `RFree`;
2. the region is active at every call to `RMalloc`; and
3. pointers that directly or indirectly point to a region are dead at the `RFree` call for that region (this corresponds to the relaxed rule R3$'$(a,b)).

All these problems are fundamentally interprocedural.

The first test above ensures that every call to `RInit` except the first is preceded by `RFree` on every path. At any point in the program, we define *RegionStatus* as being 0 or 1 depending on whether a region is inactive or active respectively. The language rules imply that *RegionStatus* is statically determinable at any point in the program. Functions that do not have any calls to `RInit` or `RFree` either directly or via a series of function calls are exceptions to this rule and are regarded as being *RegionStatus neutral*. Trusted functions are also neutral.

Our first goal is to compute, for each non-neutral function *F*, the *RegionStatus* at the entry (*RegionStatusIn[F]*) and the exit (*RegionStatusOut[F]*) of the function. *RegionStatusIn* for the topmost function in the call graph is set to be 0. For each function, we implement an intraprocedural algorithm that is essentially a data flow problem which establishes equivalence relationships between *RegionStatusIn* and *RegionStatusOut* values for the functions. For instance, the presence of a call to `RInit` or `RFree` between calls to non-neutral functions *F1* and *F2* on some path in the program fixes the values of *RegionStatusOut[F1]* and *RegionStatusIn[F2]*. Similarly if there is an `RInit` or `RFree` before any other call in a function, *RegionStatusIn* for that function is fixed. Consecutive calls to *F1* and *F2* (both non-neutral) on some path makes *RegionStatusOut[F1]* and *RegionStatusIn[F2]* equivalent. The equivalences generated are used to find the *RegionStatus* values. Any contradiction is signaled as an error by the compiler. In practice, the union-find algorithm using path compression is an efficient way to represent the equivalences. The above algorithm ensures that a single region is active at any time.

The second test above simply requires checking that all `RMalloc` call sites have a *RegionStatus* of 1. This ensures that a region is active at the call site.

The remainder of our analysis ensures that local or global pointer variables that potentially point to the heap or contain pointers that point to the heap need to be re-initialized before they are used again after an `RFree` or a call site that changes *RegionStatus* from 1 to 0. Checking pointer re-initialization is essentially an interprocedural *live variable analysis* since LLVM is in SSA form. We also check that global or local variables of aggregate types do not contain pointers, unless it can be statically proved that these pointers never point to the heap using the data structure graph. The LLVM link-time compiler provides alias information through a representation called the *data structure graph* [14], which provides points-to information connecting all disjoint memory objects (including heap, stack, globals and functions) as well as SSA pointer variables. It is computed using a fast, flow-insensitive, context-sensitive analysis. This representation directly helps identify pointer variables that could point to the heap or could contain pointers to the heap.

5. RESULTS

In this section, we first evaluate the expressiveness of the Control-C language by porting three different classes of experimental controllers (originally written in C) to the language. We believe that our example codes for each of these classes are representative of typical constructs used in such control codes. We then demonstrate that the language and the compiler are effective at detecting a number of different bugs and attacks that capture a comprehensive range of potential safety threats for the Simplex environment.

5.1 Evaluating Language Expressiveness

We ported three different classes of control applications from C to Control-C. Details of the applications are summarized in Table 1. These applications include:

1. PID (proportional integral derivative) controllers for the inverted pendulum experiment running on Simplex.
2. LQR state space controllers for the Pendubot experiment of the controls laboratory at Illinois.
3. Real-time sensor applications used in sensor networks running the TinyOS operating system [1].

The controllers for the inverted pendulum (IP) and for the Pendubot experiment are single-loop real-time control applications that control mechanical devices, but use different control algorithms. The IP controller has a single task consisting of an infinite loop that reads the status of the device, does some computation, and

Application	Platform	Size (lines)	Dynamic memory	Array accesses	Pointers	Lines changed
Inverted Pendulum	Linux	300	No	Yes	No	0
Pendubot	Windows	1300	Yes	Yes	Yes	33
TinyOS apps	Linux	300	No	Yes	Yes	0

Table 1: Control Applications Tested

sends out a control output in each iteration. The task uses simple data structures, primarily arrays, and is computationally intensive.

The Pendubot controllers are logic-based switching controllers, which are a special case of hierarchical hybrid controllers. There are essentially two or three different controllers which the supervisor switches between, depending on the state of the device. Each of the controllers is a single task similar to the IP controller task in control structure, but using different data structures. The Pendubot controllers use dynamic memory allocation to allocate some arrays before the control loop, which are freed at the end of the program.

TinyOS [1] is a component-based operating system for networked sensors. Its applications have a simple code structure that allocate most of their memory statically prior to running the application and do not use dynamic memory. The code in TinyOS applications tends to be simpler in general than typical control code applications, allowing our compiler to easily check its safety.

Each of the applications described above needed zero or minimal changes in the code in order to be ported to Control-C as shown in the last column of Table 1. We had to make the following changes to the Pendubot controller:

- Pointer arithmetic used to traverse an array had to be replaced by array index variables.
- Dynamic memory allocation needs to be region-based. This required the addition of an `RInit` and `RFree` around the boundaries of the region, converting all `malloc` calls to `RMalloc`. Also occasionally, pointers need to be re-initialized after a region free instruction if they are used subsequently.
- Unions that contain pointers are disallowed and need to be split into different variables in order to pass safety checks.

Note that the lack of annotations and enhanced types in our language greatly decreases the required changes to the source code.

We did not discover any potential security holes in any of the tested codes. This, however, is not unexpected since the codes are not malicious, and use simple data structures without complicated use of arrays and pointers. They demonstrate that the simplicity of embedded control applications with respect to the data structures enables us to guarantee safety statically. Overall, these results indicate that Control-C can be used to program a wide range of control and embedded applications.

5.2 Detecting Potential Errors and Attacks

In order to evaluate the effectiveness of the language and compiler in preventing attacks on control codes, we tested the compiler against a wide range of potential attacks that can be hidden within control code for the inverted pendulum in Simplex. Without the Control-C compiler, each of these attacks brings down the pendulum. We evaluate attacks using the following mechanisms: (1) Illegal casts, (2) Uninitialized pointers, (3) Array overflow and pointer arithmetic, and (4) Use of a pointer to access freed heap memory.

The attacks described below were constructed over time to identify a range of security threats for the Simplex environment and for use in previous demonstrations. The experiments show that all these kinds of attacks are rendered infeasible because of the normal safety mechanisms in Control-C, without requiring any special language features or compiler analysis. Furthermore, these are a fairly comprehensive set of examples in terms of the fundamental coding mechanisms that can be used to subvert safety from within an executing program. Thus, the experiments serve as a useful evaluation of the safety mechanisms designed into the language.

The code snippets shown below illustrate the mechanisms that can be used as attacks. The code snippets are from applications that run on an embedded platform with Linux 2.2.18 and control the inverted pendulum. In each of the attacks, the `killcode` array shown in Fig. 3 represents binary code for the StrongARM that executes the `kill(-1, 9)` system call, so that the code is hidden in the data area. Each attack uses a different mechanism to jump to the array address in order to execute this code. Since the Simplex environment has limited process protection, invoking this system call kills all running control tasks and brings down the pendulum.

```
char killcode[] =
    "\x55\x89\xe5\x89\xe5\xb9\x09\x00\x00
     \x00\xbb\xff\xff\xff\xff\xb8\x25\x00
     \x00\x00\xcd\x80\x89\xd3\xc3\x90";
```

Figure 3: Source for the `killcode` array

```
void controlFunction(float a, float b) {
   void (*func)();
   func = (void (*)) &killcode[0];
   func();   // jump to &killcode[0]
   ...
}
```

Figure 4: Illegal cast

```
void controlFunction(float a, float b) {
   int *ret;
   ret = (int*)&ret + 2;
   *ret = killcode;    // force return to &killcode[0]
   ...
}
```

Figure 5: Illegal Use of Pointer Arithmetic

In Fig 4, a cast from a character string to a function pointer is used to jump to the code in the `killcode` array. This cast is illegal in Control-C and the compiler rejected it during the type-checking phase. In Fig. 5, pointer arithmetic is used to overwrite the return address of the function `controlFunction` on the stack, replacing it with the address of the array `killcode`, which is executed upon return from the function. Arithmetic on pointers is illegal in Control-C and was again rejected by the type-checking phase.

Fig. 6 demonstrates the use of uninitialized pointers to compromise the system. Invocation of the function `init` initializes the stack with values such that when `func` is invoked, the local pointer variable `p` in `func` contains the address of the return address of `func` on the stack. Dereferencing `p` modifies the return address. This in turn, results in the the skipping of the instruction decrementing `'i'` after the return, leading to the overflow of `mainbuf`. This in turn overwrites the return address of `controlFunction` with

```
void func() {
  int *p;     /* contains address of return
                 address stack slot */
  (*p) += 22; /* modifies return address */
}
void init() {
  long i;
  i = &i - 2; /* Store address of return
}             * address stack slot */
void controlFunction(float a, float b) {
   int mainbuf[3], i = 4;
   init();      /* Stack initialized */
   func();             /* returns 3 lines down */
   if ((int)a % 2) {
     i -= 3;                  /* never executed */
     mainbuf[i] = killcode;  /* force return to
                                &killcode[0] */
   }
}
```

Figure 6: Illegal return via an uninitialized variable

the address of `killcode`. This use of the uninitialized pointer `(*p)` was detected and rejected by the compiler. The function `init` was rejected by the compiler because of the illegal cast from a pointer to a long integer. An uninitialized pointer however, could acquire convenient values in malicious code in many other ways and hence need to be detected.

The use of a pointer that points to a region that has been freed can be exploited in a very similar manner. The dynamically allocated memory is first initialized with convenient values which can be gotten by dereferencing a pointer to the region after the region is freed. This can, as before, be used to overflow arrays and cause the execution to jump to data areas.

The array overflow attack illustrated in Fig 7 also uses a similar technique. The function, `func` allocates an array of size 161, but overwrites two locations over the declared size, which is the return address for `func`. As before, this causes the return address of `controlFunction` to be overwritten by the address of `killcode`. The compiler detects that the out-of-bounds array reference and rejects the code.

```
void func(int z) {
   int A[161];
   for (i = 0; i <= 2*z + 1; ++i) A[i*2] += 20 ;
}
void controlFunction(float a, float b) {
  int mainbuf[3], i = 4;
  func(40);             /* returns 3 lines down */
  if ((int)a % 2) {
    i -= 3;                  /* never executed */
    mainbuf[i] = killcode;  /* force return to
                                &killcode[0] */
  }
}
```

Figure 7: Array Overflow attack

6. RELATED WORK

There have been several languages (for example Ada [11] and the Embedded Machine [10]) that specifically support real-time programming via language mechanisms that control the timing and scheduling of real-time tasks. These languages do not provide specific features for memory safety, except for a few features such as the bounded array indexing in Ada [11].

As noted in the Introduction, safe languages like Java [8], Modula-3, ML, Safe-C [2] and CCured [19] use extensive runtime checks to provide memory safety. Such checks are needed for many different properties such as array bounds checking, null pointer references, and type coercions. Some of them also rely on runtime garbage collection in ensuring the safety of pointer dereferences. These languages provide safety but are not directly suitable for real-time systems.

RT Java aims to provide a language for real-time applications while retaining the safety guarantees of Java [4]. RT Java incorporates incremental garbage collection algorithms under real-time schedulers in order to avoid the timing challenges of standard garbage collection, but these do not reduce the runtime overhead of garbage collection. In fact, RT Java has three additional flavors of dynamic memory that are not garbage collected, but which require extensive runtime checks for ensuring safety of references between the different flavors. RT Java also inherits the other runtime checking needs of standard Java such as for arrays and type coercions. RT-Java is likely to be an important technology for many real-time domains such as multimedia processing, but the runtime overheads remain a significant drawback for embedded control systems.

Fundamentally, starting with a safe language like Java and making it suitable for real-time programming presents major technical difficulties. In contrast, our approach is to take a "low-level" language (C) that has been widely used for real-time systems, and add some restrictions to provide static safety guarantees. There have been a number of other systems that provide some static safety guarantees for C-like languages, and we compare them here.

Cyclone [12, 9] uses a novel and powerful type mechanism to perform static safety checking for dynamically allocated memory. They use a more powerful region mechanism than ours, allowing arbitrary *nested* regions for the heap and separate regions for global variables and the local variables of each function. Within a function body, they restrict a pointer to not point to locations in two different regions. Unlike us, they require a number of new region annotations to expose the region accessed by a pointer, to permit more than a fixed number of regions, and to ensure that memory safety can be checked without interprocedural analysis. In contrast, we restrict an application to use a single dynamic region, restrict the usage of pointers within statically allocated aggregate structures, We rely on link-time interprocedural analysis to avoid annotations. Finally, Cyclone has to fall back on runtime checks for array bounds checks They report negligible runtime overheads for networking applications but overheads ranging from 25% to 3x for compute-intensive applications.

The Vault [6] language also uses type annotations to check the safety of memory allocation and memory accesses for memory regions statically. In fact, they use a more powerful type system that can allow many correctness requirements for dynamic resources to be encoded within the static type system and checked at compile-time; memory regions are just one application. Like Cyclone, they cannot check array accesses statically because their language mechanisms do not capture properties of arithmetic computations. Their mechanisms can be valuable for guaranteeing safety properties of system calls (which manipulate different kinds of runtime resources). In our future work, we aim to extend the Control-C language with Vault-like mechanisms to ensure the safety of system calls within networks of embedded devices.

A valuable strategy for compiler-based secure and reliable systems is Proof-Carrying Code (PCC) [17]. The benefit of PCC is that the safety checking compiler (usually a complex, unreliable system) can be untrusted, and only a simple proof checker (which can be made much more reliable) is required within the trusted code base. Fundamentally, PCC does not change what aspects of a program require static analysis and what require runtime checking

– that still depends on the language design and compiler analysis capabilities. Thus, PCC is orthogonal to our work, and could be a valuable addition to a Simplex-like environment. We envisage using PCC within Simplex, allowing our LLVM-based safety-checking compiler to be “untrusted.”

Much of the previous work on array bounds checking has been on optimizing the runtime bound checks, rather than completely eliminating them. The focus has been on moving the runtime checks out of computation intensive loops. Recently there has been some work [3, 24] on static elimination of bound checks. Our work builds on the work of ABCD [3] in that we use their constraint generation techniques within each procedure. We go beyond ABCD, however, by developing an efficient interprocedural constraint propagation algorithm. Wagner *et al.* have developed a tool for detection of buffer overrun vulnerabilities in C, based on similar techniques for generating and solving constraints. They deal with general C codes (particularly including pointer arithmetic), however, and hence decided to forego precision for scalability. Their analysis is imprecise, both in terms of generating constraints (flow-insensitive) and solving them, potentially resulting in many false-positives. In contrast, we use a more precise context-sensitive analysis and use a more rigorous constraint solver.

Finally, in a precursor project within the Simplex group, Lim et al. described how static compiler analysis (based on symbolic execution) could be used to perform static safety checking for the Simplex environment [16]. Their analysis primarily focused on array operations and did not consider pointer accesses, so that most pointer usage and any dynamic memory allocation would be rejected by their compiler. Our research aims to provide a well-defined safe language for Simplex, including support for a much wider range of control codes.

7. CONCLUSIONS AND FUTURE WORK

We have described Control-C, a programming language for secure programming of real-time control systems as well as some other embedded systems. The language is based on C but imposes key restrictions on the usage of pointers, arrays, and dynamic memory management in order to ensure that memory safety of application code can be checked at compile time, without runtime checks on memory operations, and without garbage collection. The language restrictions have been chosen to balance the needs of real-time control applications with the capabilities of static analysis. We have implemented a compiler for Control-C in a language-independent link-time system, LLVM, which permits high-level inter-procedural analyses and transformations at link-time.

We have tested our compiler on control code and embedded applications for three different experimental systems, and demonstrated that the language is expressive enough to write these applications, yet at the same time capable of detecting a fairly comprehensive set of attacks.

This work is the first step towards building a comprehensive, secure environment for dynamic upgrade of control systems and other embedded systems. Currently, our language does not detect attacks due to illegal uses of system calls, which will be particularly important for networks of embedded devices. We believe that many such errors can be detected using a combination of static techniques plus minimal run-time support. Finally, another direction would be to relax some of our current restrictions in order to support broader classes of embedded systems. While some runtime checks will then be inevitable, we can try to minimize these using sophisticated but potentially expensive compiler techniques since compilation time is typically not an important issue for embedded code.

8. REFERENCES

[1] TinyOS, a component-based OS for the Networked Sensor Regime. See web site at: http://webs.cs.berkeley.edu/tos/.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. 1994 Conf. on Prog. Lang. Design and Implementation*, Orlando, FL, June 1994.

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

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

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 13(4):451–490, October 1991.

[6] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proc. SIGPLAN '01 Conf. on Programming Language Design and Implementation*, Snowbird, UT, June 2001.

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

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Sun Microsystems, 2nd edition, 2000.

[9] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, , and J. Cheney. Region-based memory management in cyclone. In *Proc. SIGPLAN '02 Conf. on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[10] T. A. Henzinger and C. M. Kirsch. The embedded machine: Predictable, portable real-time code. In *Proc. 2002 Conf. Prog. Lang. Design and Implementation*, Berlin, Germany, June 2002.

[11] International Organisation for Standardisation. *Ada95 Reference Manual*, 1995. International Standard ISO/IEC 8652:1995.

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

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

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

[15] C. Lattner and V. Adve. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.

[16] S. Lim, K. Lee, and L. Sha. Ensuring integrity and serivce availability in a web based control laboratory. *To appear in Journal of Parallel and Distributed Computing Practices*.

[17] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[18] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

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

[20] W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, Aug. 1992.

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

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

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

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