SAFECode: Enforcing Alias Analysis for Weakly Typed Languages *

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

Abstract

Static analysis of programs in weakly typed languages such as C and C++ is generally not sound because of possible memory errors due to dangling pointer references, uninitialized pointers, and array bounds overflow. We describe a compilation strategy for standard C programs that guarantees that aggressive interprocedural pointer analysis (or less precise ones), a call graph, and type information for a subset of memory, are never invalidated by any possible memory errors. We formalize our approach as a new type system with the necessary run-time checks in operational semantics and prove the correctness of our approach for a subset of C. Our semantics provide the foundation for other sophisticated static analyses to be applied to C programs with a guarantee of soundness. Our work builds on a previously published transformation called Automatic Pool Allocation to ensure that hard-to-detect memory errors (dangling pointer references and certain array bounds errors) cannot invalidate the call graph, points-to information or type information. The key insight behind our approach is that pool allocation can be used to create a run-time partitioning of memory that matches the compile-time memory partitioning in a points-to graph, and efficient checks can be used to isolate the run-time partitions. Furthermore, we show that the sound analysis information enables static checking techniques that eliminate many run-time checks. Our approach requires no source code changes, allows memory to be managed explicitly, and does not use meta-data on pointers or individual tag bits for memory. Using several benchmarks and system codes, we show experimentally that the run-time overheads are low (less than 10% in nearly all cases and 30% in the worst case we have seen). We also show the effectiveness of static analyses in eliminating run-time checks.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages

General Terms Reliability, Security, Languages

Keywords Compilers, programming languages, alias analysis, region management, automatic pool allocation.

* This work is supported in part by the NSF Embedded Systems program (award CCR-02-09202), the NSF Next Generation Software Program (award CNS 04-06351), and an NSF CAREER award (EIA-0093426).

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

1. Introduction

Alias information, type information, and call graphs are the fundamental building blocks for many kinds of static analysis tools, including model checkers and error checking tools. For programs written in weakly typed languages, however, these fundamental building blocks may not be valid if the program performs any illegal memory operations such as array bound violations, dangling pointer dereferences, and references using uninitialized pointers, because these unsafe operations can overwrite memory locations in ways not predicted by the compiler. This means that even tools that aim to provide sound results with no false negatives [24, 10] cannot guarantee that they do so. In fact, software validation tools usually assume that such memory corruption cannot occur, e.g. they assume malloc always returns fresh memory (so dangling pointer references cannot occur) and that memory allocations are logically infinitely apart (so a buffer overflow cannot trample any other allocation). This problem is potentially important because many software validation tools today are used to detect security vulnerabilities or identify logical errors in important system software.

Unfortunately, it has proven extremely expensive to detect important classes of unsafe memory operations for a weakly typed language using static analysis, run-time checks or a combination of both [2, 39, 27, 34, 32]. All of these approaches have overheads that are prohibitively high for production use (e.g., 2x-11x). Furthermore most of these use heuristic techniques to detect certain errors, especially dangling pointer errors, and do not guarantee that all such errors will be detected.

An alternative approach is to use strongly typed systems that closely match the C type system, e.g., CCured [33] or Cyclone [20, 25]. The strong safety guarantees of these systems are technically attractive but they are obtained by disallowing explicit memory deallocation in general ([25] allows explicit deallocation in some restricted cases) and relying on automatic memory management. The adoption of automatic memory management for existing C software is likely to be slow for several reasons. First, it can take significant effort to tune legacy C programs to reuse memory effectively in a managed environment. Second, system and embedded software often have stringent requirements for performance, memory consumption, real-time constraints, and even power constraints. C has been widely used for such software partly because of the control it gives over performance and memory consumption. For these reasons (and because of the possible manual effort required to port programs to these languages), existing C and C++ applications may be slow to adopt such languages and many may not do it at all.

1.1 Overview of our approach

In this paper, we describe a novel, automatic approach an ordinary compiler can use to ensure three key analysis results — namely, a points-to graph, a call graph, and available type information —

are sound, i.e., will not be invalidated by any possible memory errors, even undetected errors such as dangling pointer dereferences. We achieve this by defining new operational semantics based on the analysis results. We give a formal proof of soundness of our approach for a subset of C.

Our solution builds on a previously published transformation we call *Automatic Pool Allocation* [31]. Automatic Pool Allocation uses the results of a pointer analysis to partition heap memory into fine-grain pools while retaining explicit deallocation of individual objects within pools, i.e., it partitions the heap but does not perform automatic memory management. The transformation was developed and used for optimizing memory hierarchy performance. The transformation essentially provides a run-time partitioning of the heap *that corresponds directly to the partitioning of memory in the points-to graph.*

The primary contribution of this work is to show how Automatic Pool Allocation can be used to enforce the validity of a given points-to graph despite potential memory errors, and to do so efficiently. This work is based on two key observations. First, by partitioning memory (at least) as finely as the points-to graph, we can check efficiently that a pointer does not reference a memory object that is not in its predicted points-to set. Second, many pools have a type-homogeneity property that allows us to eliminate many or most of these run-time checks. Furthermore, the type-homogeneity property also allows us to statically ensure that dangling pointer errors will not cause any unexpected type violation.

There are four technical challenges that we solve in this work in order to use Automatic Pool Allocation for guaranteeing that the static analysis results are correct:

- We formalize the necessary properties of pool allocation as a new type system and the necessary run-time checks in an operational semantics so that we can prove the correctness of the overall system. In an accompanying technical report [13], we give a formal proof of correctness for a subset of C that includes all important root causes of memory errors: dangling pointers, arbitrary casts and type mismatches, uninitialized variables, and array bounds violations.
- Second, pool allocation does not prevent dangling pointer references to freed memory. We show how to exploit type homogeneous pools to ensure that such dangling references do not cause any unexpected type violations in these pools. We have used this insight in previous work on enforcing memory safety, but only for a subset of C without non-type safe constructs and we did not prove its soundness [15].
- Non-type-safe constructs in C (e.g., many pointer casts, unions, and varargs function calls) or pointer analysis imprecision may produce non-type-homogeneous pools. We show how to use run-time checks to enforce isolation of such pools from each other and from type homogeneous pools. We also need additional run-time checks to detect other memory errors such as uninitialized references and array bounds violations.
- Finally, we show that we can use *sound* static analyses that exploit the points-to graph and call graph to *safely* optimize away many of the run-time checks and other run-time overheads introduced by SAFECode. We also give an example to show how a different static analysis tool (ESP [10]) could benefit from our approach.

Our approach has several practical strengths (discussed later in Section 9) and three key limitations. The first two limitations are the requirements that the pointer analysis be flow-insensitive and unification-based. We believe our approach can be extended to enforce non-unification based pointer analysis as well, by adding meta-data to every pointer that may target multiple pools. However, the requirement of flow-insensitivity may be much harder to relax (though, as discussed in Section 6.2, sound flow-sensitive techniques can be implemented on top of our approach.) The third limitation is that our analysis does not detect all memory errors; it only guarantees sound pointer analysis and call graph semantics with low run-time overhead in the presence of memory errors.

We have implemented our techniques in a system we call SAFECode - Static Analysis For safe Execution of Code - using the LLVM compiler infrastructure [30]. Our system handles nearly the full generality of C, except programs with "manufactured addresses." We show experimentally using three groups of programs (Olden, Ptrdist, and three daemons) that the run-time overheads of SAFECode are close to 0 for most programs and less than 30% in all cases we have tested. We also show that the static analyses, whose correctness relies on alias analysis guaranteed by SAFECode, are important for achieving these low overheads.

The next section describes the language and analysis representations we assume as our inputs. Sections 3 describes our overall approach, type system, and operational semantics for a subset of C. Section 4 discusses the extensions to the type system to handle the full generality of C programs. Section 5 describes our implementation (SAFECode) of the run-time system. Section 6 describes static analyses that benefit from SAFECode. Section 7 presents our experimental evaluation of SAFECode. Section 8 discusses related work and section 9 concludes with directions for future work.

2. Assumptions and Background

The inputs to our approach are:

1. a program written in C;
2. The results of a flow-insensitive, field-sensitive, unification-based pointer analysis on that program. As explained below, this includes both points-to information and type information for some subset of memory objects. The analysis may use various forms of context-sensitivity (see below).
3. A call graph computed for the program.

Our goal is to enforce the correctness of these analyses for all executions of the program. We do not concern ourselves with how these analysis results are actually computed; we only assume that these are given in the format described below. In our implementation, we use an analysis called Data Structure Analysis (DSA) [29], a context-sensitive, field sensitive, unification based algorithm to compute both the pointer-analysis and the call graph.

We include type information as part of the points-to representation because a pointer analysis can infer such information in a weakly typed language. For example, DSA does this by checking if all pointers to a particular "points-to set" (explained below) are used or indexed consistently as one type $\tau*$ (where uses do not include casts). If so, DSA marks the type of objects in the set to be τ and otherwise the type "*Unknown*," explained below.

Although our implementation of SAFECode supports all of C, we use a subset of C as the source language in this paper to better illustrate our main ideas. This language, shown in Figure 1, includes most sources of potential memory errors in weakly typed languages including:

(**P1**) dangling pointers to freed heap memory,

(**P2**) array bound violations,

(**P3**) accesses via uninitialized pointers, and

(**P4**) arbitrary cast from an int type to another pointer type and subsequent use.

The remaining language features of C, namely, structures, functions, stack and global allocations, and function pointers, produce

Var			$x \quad y$
NodeVar			$\boldsymbol{\rho}$
Types	τ	::=	int | char | $\boldsymbol{Unknown}$
			| $\tau \ * \boldsymbol{\rho}$
Statements	S	::=	ϵ | S; S | x = E | store E, E
			| storec E, E | free(E, E)
			| **associate($\boldsymbol{\rho}$, $\boldsymbol{\tau}$)**
Expressions	E	::=	Var | V | E op E | load E
			| loadc E | cast E to τ
			| malloc (x, E) | &E[E])
Value	V	::=	$Uninit$ | Int

Figure 1. The input language (we omit structs, stack allocations, globals, functions, and function pointers here; they are discussed in Section 4). The constructs shown in bold are necessary to represent the pointer analysis as a part of the input program.

two other kinds of memory errors: dangling pointers to a stack frame after a function returns and illegal indirect calls. The language extensions for the same are discussed later in Section 4.

In our input language, we include `int` (4-byte) and `char` (1-byte) as primitive data types, and use distinct load and store operations for these types (e.g., load E for loading `int`s and loadc E for loading chars). The `cast, malloc, free` operations are similar to those in C. &E[E] is for pointer arithmetic of C.

2.1 Pointer Analysis Representation

Intuitively, pointer analysis representation can be thought of as a storage-shape graph [36, 26] (also referred to as a points-to graph), where each node represents a set of dynamic memory objects (a "points-to set") and distinct nodes represent disjoint sets of objects. Pointers pointing to two different nodes in the graph are not aliased. We assume there is one points-to graph per function, since this allows either context-sensitive or insensitive analyses. Figure 2 (a) shows an example program in source language and Figure 2 (b) the associated storage-shape graph.

We assume that the input pointer analysis is encoded as type attributes within the program, using a type system analogous to Steensgaard's [36]. Each points-to graph node is encoded as a distinct type (although we continue to refer to nodes below). The input to our approach is a program in this language. Figure 2 (c) shows the running example in our input language. Each pointer in this language has an extra attribute, ρ, which intuitively corresponds to the node it points to in the points-to graph. For example, in Figure 2(c), the type of `y` is `int*r2`, denoting that it points to objects of node `r2` in the points-to graph. The statement `associate`(ρ, τ) associates node ρ of the graph with type τ, denoting that the node ρ contains objects of type τ. If τ is a pointer type, say, $\tau' * \rho'$, then `associate`$(\rho, \ \tau' * \rho')$ directly encodes a "points-to" edge from node ρ to node ρ'. These `associate` statements are typically listed at the beginning of each function. Note that there can be only a single target node for each variable (or field of pointer type), which restricts the input to a unification-based pointer analysis.

Memory that is used in a type-inconsistent manner, e.g., due to unions or casts in C, must be assigned type $Unknown$ (this is verified by our type checker). $Unknown$ is interpreted as an array of chars. In the running example, the target of z (node `r3`) has type $Unknown$ because this memory is accessed both as an `int` and as an `int**`. Distinct array elements (due to $Unknown$ or an actual array) are not tracked separately.

In the absence of `frees` and other memory errors, we can check that this program encodes the correct aliasing information by using typing rules similar to Steensgaard's. We do not give those rules here as our approach described in Section 3 is stronger and subsumes this checking; we not only check that the static aliasing is correct but we also enforce it in the presence of memory errors.

2.2 Background on Automatic Pool Allocation

Given a program containing explicit `malloc` and `free` operations and the pointer analysis information above, Automatic Pool Allocation transforms the program to segregate data into distinct pools on the heap [31]. Pool allocation creates a distinct pool, represented by a pool descriptor variable (also known as a pool handle), for each points-to graph node representing heap objects in the program. For a points-to graph node with $\tau \neq Unknown$, the pool created will only hold objects of type τ (or arrays thereof). i.e., the pools will be *type-homogeneous*. We refer to such *type-homogeneous* pools as **TK** (type known) and all others as **TU** (type unknown). Calls to `malloc` and `free` are rewritten to call new functions `poolalloc` and `poolfree`, passing in the appropriate pool descriptor.

In order to minimize the lifetime of pool instances, pool allocation examines each function and identifies points-to graph nodes whose objects are not reachable via pointers after the function returns. This is a simple reachability analysis on the points-to graph for the function. The pool descriptor for such a node is created on function entry and destroyed on function exit so that a new pool instance is created every time the function is called. For other nodes, pool allocation adds new arguments to the function to pass in the pool descriptor from the caller. Finally, each function call is rewritten to pass any pool descriptors needed by any potential callee.

In our previous work, we have used Automatic Pool Allocation to improve memory hierarchy performance [31] and to enforce memory safety without automatic memory management in a type-safe subset of C [15]. The current work is the first to consider how automatic pool allocation can be used to enforce the correctness of a points-to graph, call graph and type information.

3. Type System

3.1 Overview

We first give an informal overview of our approach, focusing on four key insights we exploit in this work. The first two are new in the current work while the other two are borrowed from our previous work on memory safety for a type-safe subset of C [15].

The goal of our work is to ensure that memory errors (e.g., dangling pointer references after a free, array bounds violations, etc.) do not invalidate the points-to information, call graph, or type information computed by the compiler. The major challenge is enforcing points-to information; type information follows directly from this. The call graph is simply checked explicitly at each indirect call site (See Section 4 for a discussion on eliminating some of the run-time checks at indirect call sites).

Note that a node in a points-to graph (or the storage shape graph) is just a static representation of a set of dynamic memory objects. If these memory objects are scattered about in memory (as is usually the case), it is prohibitively expensive to check that a pointer actually points to a memory object corresponding to its target node (i.e., has not been corrupted by some memory error). As noted earlier, however, our transformation called *Automatic Pool Allocation* partitions the heap into regions based on a points-to graph [31]. This leads us to the following new insight that is the key to the current work:

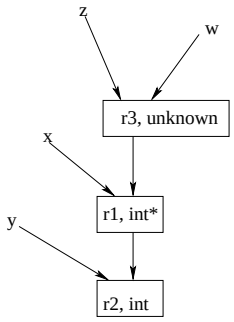
[**Insight1**]: *If memory objects corresponding to each node in the points-to graph are located in a (compact) region of the heap, we could check efficiently at run-time that the target of a pointer is a valid member of the compile-time points-to set for that pointer, i.e., that alias analysis is not invalidated.*

Note that this insight relies on the property that unaliasable memory objects are not allocated within the same region, which is not usually guaranteed by previous region-based systems [38, 20].

Non-heap (i.e., global and stack) objects may be in the same or different points-to sets as heap objects. We can simply include such

```
int **x, *y, *z, ***w, u;
x = (int **) malloc(4);
y = (int *)malloc(4);
z = (int *) malloc(4);
...
store y, x  // equivalent of *x = y
store 5 ,y
free(z) ; // creates a dangling pointer
store 10, z;
...
u = load z;  // equivalent of u = *z;
...
w = cast z to (int ***);
store x, w;
```

(a)



(b)

```
associate(r1, int * r2);
associate(r2, int);
associate(r3, Unknown);
int *r2 *r1 x;
int *r2 y;
int *r3 z; *r2*r1*r3 w;  int u;
x = malloc(4);
y = malloc(4);
z = malloc(4);
...
store y, x; store 5, y;
free(z);      // dangling pointer still exists
store 10, z ;
...
u = load z ;
...
w = cast z to (int *r2*r1*r3);
store x, w ;
```

(c)

Figure 2. (a) Original program, (b) its points-to graph, and (c) program in our type system

objects in the set of address ranges for the appropriate pool (but many stack objects can be handled more efficiently as described in Section 4). Overall, the operation `poolcheck`(ph, A, o) verifies that the address, A, is contained within the set of memory ranges assigned to pool, ph, and has the correct alignment for the pool's data type (or for the field at offset o if $o \neq 0$).

Even with the above partitioning of memory, checking every pointer dereference (or every pointer definition) would be prohibitively expensive. The second insight allows us to eliminate a large number of the run-time checks:

[**Insight2**]: *Any initialized pointer read from an object in a TK region or from an allocation site, will hold a valid address for its target region. All other pointers, i.e., pointers derived from indexing operations, and pointers read from TU regions (including function pointers), need run-time checks before being used.*

Intuitively, in the absence of dangling pointer errors and array indexing errors, an initialized pointer obtained from a TK region will always be valid; it cannot have been corrupted in an unpredictable way e.g. via arbitrary casts and subsequent stores (it would then be obtained from a TU region).

Uninitialized pointers and array indexing errors are addressable via run-time checks. Dangling pointer references, however, are difficult to detect in general programs, and we do not attempt to detect or prevent such errors. Instead, we ensure that such errors do not invalidate the results of alias analysis, by exploiting two ideas that we also used in previous work on enforcing memory safety for a type-safe subset of C [14, 15]:

[**Insight3**]: *In a TK (type-homogeneous) region, if a memory block holding one or more objects were freed and then reallocated to another request in the same region with the same alignment, then dereferencing dangling pointers to the previous freed object cannot cause either a type violation or an aliasing violation.*

Essentially, we make sure that, if a dangling pointer to freed memory points into a newly allocated object, the old and new objects have the same static type and that any pointers they contain have identical aliasing properties. Thus loads or stores using the dangling pointers may give unexpected results but cannot trample memory outside the expected pool.

This principle allows free memory to be reused *within* the same region (unlike other region-based languages, which either disallow such reuse [38] or allow it only in restricted cases [25, 37]). For reuse across regions, as we noted in our previous work [15], Automatic Pool Allocation already provides us a solution:

[Insight4]: *We can safely release the memory of a region when there are no reachable pointers into that region.*

This gives us a way to release memory to the system. Since Automatic Pool Allocation already binds the life times of regions (using escape analysis), we can arrange for memory to be released at the end of a region's life time.

Finally, in order to prove the correctness of our approach, we formalize the key properties of our regions by extending the previous type system encoding points-to information (described in Section 2) in two ways: (1) to encode regions corresponding to points-to sets, with allocation and deallocation out of these regions; and (2) to encode information about region lifetimes. The type system is designed to be mostly statically checkable for the correctness of encoded types (i.e. the points-to relations, lifetimes, and the call graph). We borrow a key idea from Tofte and Talpin's work on regions for ML [38] to simplify the type system, namely, we restrict region lifetimes to be lexically scoped (others have shown that this is not strictly necessary [1, 20]).

3.2 Syntax

Figure 3 gives the syntax of the language in our type system. This syntax, which forms the input to our type checker, includes new constructs for encoding region handles, region lifetimes, region allocation and deallocation, and separate versions of `load/store` that require run-time checks. The `associate` statement of Figure 1 is now transformed to the `poolinit` statement along with a lexical scope indicating where the association is valid, essentially creating a lifetime for the corresponding region. For example, statement `poolinit`(ρ, τ) $x_\rho\{$ S $\}$, creates a region named ρ that can hold objects of type τ, with the handle x_ρ. Our typing rules, described in Section 3.3, make it illegal to store an object of type other than τ in this region. The type of the region handle x_ρ is handle(ρ, τ). Notice that regions in our system are nested and a region can only contain objects of one type (although this type may have to be *Unknown* for some regions). The lexical scoping, along with region attributes for pointers, allow the type checker to ensure that an object in a region cannot be accessed outside the lifetime of the region. Although this seems to disallow cycles in points-to graph, extensions for handling them are straightforward; we support creating multiple regions at the same lexical level (not present in the syntax here) and it can be used to create regions for all the nodes in a cycle at once (discussed further in the technical report [13]).

Calls to `malloc` and `free` in the input program are replaced by calls to `poolalloc` and `poolfree`. `poolalloc` takes in a handle

RegionVar			ρ
Var			$x \quad y$
Types	τ	::=	int | char | $Unknown$
			| $\tau \ * \ \rho$ | handle(ρ, τ)
Statements	S	::=	ϵ | S; S | x = E | store E, E
			| storeToU x, E, E | storec E, E
			| storecToU E, E | poolfree(E, E)
			| poolinit(ρ, τ) x { S }
			| pool{S}pop(ρ)
Expressions	E	::=	var | V | E *op* E | load E
			| loadFromU x, E | loadc E
			| loadcFromU E | cast E to τ
			| poolalloc (x, E) | (x,&E[E])
			| castint2pointer x,E to τ
Value	V	::=	$Uninit$ | Int | region(ρ)

Figure 3. Our syntax

```
int *r2 *r1 x;
int u, *r2 y
int *r3 z;
int *r2*r1*r3 w;
poolinit(r2, int) r2handle {
  poolinit(r1, int *r2) r1handle {
    poolinit(r3, Unknown) r3handle {
      x = poolalloc(r1handle, 1);
      y = poolalloc(r2handle, 1);
      z = poolalloc(r3handle, 1);
      store y, x;        store 5, y;
      poolfree(r3handle,z); //dangling pointer exists
      storeToU r3handle, 10, z ;
      ...
      u = loadFromU r3handle, z ;
      ...
      w = cast z to (int *r2*r1*r3);
      storeToU r3handle, x , w ; //type checks as region of r3
      ...                        //is Unknown
} } }
```

Figure 4. Running example in our type system

to the region as an argument and allocates an object (or an array of objects) out of the region. The type of an allocated object (or array element) is the type associated with the region. The `poolfree` statement frees a memory object and releases the memory back to the region. $Uninit$ essentially represents the NULL value in C. The `castint2ptr, loadFromU, storeToU` are versions of `cast, load, store` that require various run-time checks. Other than Uninitialized pointer checks, the only operations that require a run-time check are those that take in a pool handle as an argument.

Everything else in the syntax including pool{(S)}pop(ρ) , region(ρ) are not part of the source language but needed for operational semantics and are described in section 3.4.

The Figure 4 shows the running example in this new syntax. The `storeToU` and `loadFromU` operations are versions of `store` and `load` that need run-time checks. The `associate` is now replaced by `poolinit`, binding the lifetime of the pools.

We use Automatic Pool Allocation transformation to take an input program including the pointer analysis annotations described in Section 2, and produce a program with region types and region allocation.

3.3 Typing rules

The type system is expressed by the following three judgments: $C \vdash e : \tau$ (for expression typing) , $C \vdash S$ (for statement typing), $C \vdash \tau$ (for type typing).

$$\text{(SS0)}\ \frac{C \vdash \tau \quad \Gamma(x) = \tau}{C(=\Gamma, \Delta) \vdash x : \tau} \qquad \text{(SS1)}\ \frac{}{C \vdash n : \text{int}}$$

$$\text{(SS2)}\ \frac{C \vdash e1 : \texttt{int} \quad C \vdash e2 : \texttt{int}}{C \vdash e1 \ op \ e2 : int}$$

$$\text{(SS3)}\ \frac{C \vdash \tau}{C \vdash Uninit : \tau}\ \tau \neq \text{handle}(\rho', \tau')$$

$$\text{(SS4)}\ \frac{C \vdash e : \tau * \rho \quad C \vdash \rho : \tau \quad \tau \notin \{Unknown, \texttt{char}\}}{C \vdash \text{load } e : \tau}$$

$$\text{(SS4char)}\ \frac{C \vdash e : \texttt{char} * \rho \quad C \vdash \rho : \texttt{char}}{C \vdash \text{loadc } e : \texttt{char}}$$

$$\text{(SS5)}\ \frac{\begin{array}{c} C \vdash e : \tau * \rho \quad C \vdash \rho : Unknown \\ C \vdash x : \text{handle}(\rho, Unknown) \end{array}}{C \vdash \text{loadFromU } x, e : \texttt{int}}$$

$$\text{(SS5char)}\ \frac{\begin{array}{c} C \vdash e2 : \tau * \rho \quad C \vdash \rho : Unknown \\ C \vdash x : \text{handle}(\rho, Unknown) \end{array}}{C \vdash \text{loadcFromU } x, e2 : \texttt{char}}$$

$$\text{(SS6)}\ \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e : int}{C \vdash \text{poolalloc}(x, e) : \tau * \rho}$$

$$\text{(SS7)}\ \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e : int}{C \vdash \text{castint2ptr } x, e \text{ to } \tau * \rho : \tau * \rho}$$

$$\text{(SS8)}\ \frac{C \vdash \tau' \quad C \vdash e : \tau * \rho}{C \vdash \text{cast } e \text{ to } \tau' * \rho : \tau' * \rho}$$

$$\text{(SS9)}\ \frac{\begin{array}{cc} C \vdash \rho : \tau & C \vdash x : \text{handle}(\rho, \tau) \\ C \vdash e2 : \tau * \rho & C \vdash e3 : int \end{array}}{C \vdash x, \&e2[e3] : \tau * \rho}$$

$$\text{(SS10)}\ \frac{C \vdash e : \tau}{C \vdash \text{cast } e \text{ to } \texttt{int} : int}\ \tau \neq \text{handle}(\rho, \tau')$$

Figure 5. Expression typing judgments

In these judgments C, the typing context, is a pair of typing environments $(\Gamma; \Delta)$ where Γ is a map between variable names and their types (built up using the variable declarations) and Δ is a map between region names and the type of objects stored in the region (built up using `poolinits`). We present the typing rules for our language in Figures 5, 6, and 7.

While many of the type rules are similar to those of C, some type rules are unique to our approach and require further explanation. (SS4) and (SS14) type loads/stores using pointers to type consistent memory (TK pools). They check that the type of the objects in the pool matches the type of the pointer operand. (SS5) is for loads using pointers to untyped $Unknown$ memory (TU pools); note that we get back an int. (SS7) allows a cast from int to pointer type, As discussed later in the operational semantics, such a cast requires a run-time check to make sure that the pointer is of the right type in the right pool. This coupled with (SS5) above enables loading pointers from TU pools safely. (SS15) types stores to $Unknown$ memory. (SS8) types a cast from a pointer to a region to another pointer pointing to the same region. This helps in supporting arbitrary casts of pointer types as long as they have the same region attribute, *without* requiring run-time check; note that (SS4) and (SS14) require that a pointer be cast back to the type of objects in the region before use. (SS17) is for creating a region using poolinit; we add the region variable and the handle to the typing context before checking the body of the `poolinit`. (SS6) gives a type for the memory objects allocated in a pool. (SS16) frees objects only when they belong to the appropriate pool.

$$\text{(SS11)}\ \frac{}{C \vdash \epsilon} \qquad \text{(SS12)}\ \frac{C \vdash s1 \quad C \vdash s2}{C \vdash s1; s2}$$

$$\text{(SS13)}\ \frac{C \vdash x : \tau \quad C \vdash e : \tau}{C \vdash x = e}$$

$$\text{(SS14)}\ \frac{\begin{array}{l} C \vdash \rho : \tau \quad C \vdash e1 : \tau * \rho \\ C \vdash e2 : \tau \quad \tau \notin \{Unknown, \texttt{char}\} \end{array}}{C \vdash \text{store } e2, e1}$$

$$\text{(SS14char)}\ \frac{C \vdash \rho : \texttt{char} \quad C \vdash e1 : \rho * \texttt{char} \quad C \vdash e2 : \texttt{char}}{C \vdash \text{storec } e2, e1}$$

$$\text{(SS15)}\ \frac{\begin{array}{ll} C \vdash \rho : Unknown & C \vdash e1 : \tau * \rho \\ C \vdash e2 : \tau & C \vdash x : \text{handle}(\rho, Unknown) \end{array}}{C \vdash \text{storeToU } x, e2, e1}$$

$$\text{(SS15char)}\ \frac{\begin{array}{ll} C \vdash \rho : Unknown & C \vdash e1 : \tau * \rho \\ C \vdash e2 : \texttt{char} & C \vdash x : \text{handle}(\rho, Unknown) \end{array}}{C \vdash \text{storecToU } x, e2, e1}$$

$$\text{(SS16)}\ \frac{C \vdash \rho : \tau \quad C \vdash x : \text{handle}(\rho, \tau) \quad C \vdash e2 : \tau * \rho}{C \vdash \texttt{poolfree}(x, e2)}$$

$$\text{(SS17)}\ \frac{\begin{array}{c} C \vdash \tau \quad \Gamma[x \mapsto \text{handle}(\rho, \tau)], \Delta[\rho \mapsto \tau] \vdash s \\ x \notin \Gamma \text{ and } \rho \notin \Delta \end{array}}{C(= \Gamma, \Delta) \vdash \texttt{poolinit}(\rho, \tau) x\{s\}}$$

Figure 6. Statement typing judgments

$$\text{(SS18)}\ \frac{}{C \vdash \texttt{int}, \texttt{char}} \qquad \text{(SS19)}\ \frac{}{C \vdash Unknown}$$

$$\text{(SS20)}\ \frac{\Delta(\rho) = \tau}{C \vdash \rho : \tau} \qquad \text{(SS21)}\ \frac{\vdash \rho : \tau}{C \vdash \tau * \rho}$$

$$\text{(SS22)}\ \frac{\Delta(\rho) = Unknown}{C(= \Gamma, \Delta) \vdash \tau * \rho} \qquad \text{(SS23)}\ \frac{C \vdash \rho : \tau}{C \vdash \text{handle}(\rho, \tau)}$$

Figure 7. Well formed types

3.4 Operational semantics

The operational semantics rules for our language provide a formal basis for reasoning about program behavior even in the presence of problems P1-P4. They essentially describe the run-time checks needed to enforce the correctness of alias analysis. The rules are listed in Figures 9 and 10.

Figure 8 lists the environments necessary to describe the operational semantic rules. The rules are described as a small-step operational semantics, $\longrightarrow_{expr}$ for expressions and $\longrightarrow_{stmt}$ for statements. Each program state is represented by (VEnv, L, es) where VEnv is the variable environment (partial map holding the values of variables), L is the partial map of live regions and the corresponding store, and es is an expression or statement in the program. H, the system heap, contains the memory addresses not in use by the program. H is a part of the program state but not included in the notation for the sake of brevity. A program state (VEnv, L, es) becomes (VEnv', L', es') if any of the semantic rules allow for it. The expression in the box, if any, is a run-time check that is executed before the corresponding rule. If the run-time check fails, then the program state becomes specially designated Error state.

We assume that region(ρ) is the handle for a region named ρ. A region (see Figure 8) is defined as a tuple { F ; RS }: F is a list of freed memory locations within the region, and RS (the region store) is a partial map between memory addresses and their values.

Briefly, the four memory errors **P1-P4** listed in Section 2 are solved as follows. **P1** is solved using the type homogeneity principle, explained previously. This is implemented by rules **R14, R34** and the static typing rules that check operations on pointers to known-type pools. We detect problem **P2** using the run-time check on rule **R40**. To detect **P3**, we initialize all newly created memory and all local variables to $Uninit$ and check $Uninit$ pointers via rules **R6, R14, R23**. Issue **P4** is detected using **R31**.

VarEnv	VEnv	:	Var $\longrightarrow$ Value
Region	R	::=	{ F ; RS }
RegionStore	RS	:	Int $\longrightarrow$ Value
FreeList	F	::=	$\phi \mid aF$
LiveRegions	L	:=	RegionVar $\longrightarrow$ RS
SystemHeap	H	$\subseteq$	Int_{32}

Figure 8. Environments for operational semantics

Below we describe in more detail the rules that are unique to our approach. **(R15,R17)**: Evaluating `poolinit` creates a new region, sets the free list to be empty, and evaluates the body inside the syntactic construct pool{S}pop(ρ). This construct identifies when the region is to be deallocated, i.e., when the body (S) becomes empty. This is performed by rule R17. **(R6)**: Performs a store via a type-consistent pointers, after checking that v_1 is not $Uninit$. $update(L, v1, v2)$ just updates the memory location $v1$ with value $v2$. Loads via type consistent memory have a similar check for uninitialized pointers. **(R10)**: Performs a store via a pointer to $Unknown$ memory, after checking that the pointer value legally allows storing of a 4-byte value (an int). Note that our proof below guarantees that $v \in$ Dom(L[ρ].RS), so at run-time it is enough to check for the open interval $(v_1,\ v_1\ +\ 3]$. **(R14)**: Frees an object from region, ρ, and adds it to the free list F of the same region. **(R34)**: Returns a previously freed location from the free list. Together with **R14**, this implements the type homogeneity principle to make error **P1** harmless. **(R35)**: For a poolalloc, when the free list is empty, this requests fresh memory from the system. `poolalloc` aborts if it cannot allocate requisite memory. **(R31)**: A cast from int to another pointer type is always checked at run-time using a `poolcheck`, i.e., we check that the value is a properly aligned address in the appropriate pool for the pointer type, and if not, we abort. This detects problem **P4**. **(R40)**: For array indexing, we check that the resultant pointer after the arithmetic always points to the same pool as the source pointer at the proper alignment. These checks are not exact array bounds checks but a much coarser check for the pool bounds. This means some array bound violations may go undetected.

The complete list of run-time checks in our system are the checks in the boxes in Figures 9 and 10 along with checks on casts from integer to function pointers. Note that the `poolcheck` operation described earlier in this section is exactly the check given in **R31**. None of the run-time checks require any metadata on individual pointer variables (usually required for precise array bounds checks) or runtime tag bits on any memory locations (usually required for RTTI or to track legal pointer values). The only metadata we require at runtime are available in the pool descriptor (handle), which is known at compile time.

3.5 Soundness proof

The proof of soundness is composed of two "invariant preservation" theorems — one for expressions and one for statements of the program. Since we have not included control flow in our formalization, all evaluations of expressions and statements terminate.

A detailed proof of our technique is included in a separate technical report [13]. Here we just summarize the important invariants that our approach maintains at each step of the operational semantics and state the soundness theorem for statements.

First, for an environment (VEnv, L), we define $||\tau||_{(VEnv,L)}$ to be as follows:

R1 $\dfrac{\text{(VEnv, L, S1)} \longrightarrow_{stmt} \text{(VEnv', L', S1')}}{\text{(VEnv, L, S1 ; S2)} \longrightarrow_{stmt} \text{(VEnv', L', S1' ; S2)}}$

R2 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, x = E)} \longrightarrow_{stmt} \text{(VEnv', L', x = E')}}$

R3 $\text{(VEnv, L, x = } v_1\text{)} \longrightarrow_{stmt} \text{(VEnv[x} \mapsto v_1\text{], L, } \epsilon\text{)}$

R4 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, store/storec E, E}_2\text{)} \longrightarrow_{stmt} \text{(VEnv', L', store E', E}_2\text{)}}$

R5 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, store/storec } v\text{,E)} \longrightarrow_{stmt} \text{(VEnv', L', store } v\text{, E')}}$

R6 $\text{(VEnv, L, store/storec } v_2\text{, } v_1\text{)} \longrightarrow_{stmt} \text{(VEnv, update(L, } v_1, v_2\text{), } \epsilon\text{)}$ $\boxed{(v_1)! = Uninit}$
where
update(L, v_1, v_2) := L' $\cup$ {(ρ, { R.F ;R.(RS[$v_1 \mapsto v_2$]) }) } if $\exists \rho \in$ Dom(L) s.t. L = L' $\cup$ {(ρ,R)} and $v_1 \in$ Dom(R.RS)
L else

R7 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, storeToU/storecToU E, E}_2\text{, E}_3\text{)} \longrightarrow_{stmt} \text{(VEnv', L', storeToU/storecToU E', E}_2\text{, E}_3\text{)}}$

R8 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, storeToU/storecToU } v_1\text{, E, E}_3\text{)} \longrightarrow_{stmt} \text{(VEnv', L', storeToU/storecToU } v_1\text{, E', E}_3\text{)}}$

R9 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, storeToU/storecToU } v_1\text{, } v_2\text{, E)} \longrightarrow_{stmt} \text{(VEnv', L', storeToU/storecToU } v_1\text{, } v_2\text{, E')}}$

R10 $\text{(VEnv, L, storeToU region(}\rho\text{), } v_2\text{, } v_1\text{)} \longrightarrow_{stmt} \text{(VEnv, update(L, } v_1, v_2\text{, 4) }\}\text{], } \epsilon\text{)}$ $\boxed{(v_1, v_1 + 3] \in \text{Dom(L[}\rho\text{].RS)}}$
where
update(L, v_1, v_2, 4) := L' $\cup$ {(ρ, { R.F ;R.(RS[$v_1 \mapsto$byte(v_2, 3)][(v_1+1) $\mapsto$byte(v_2, 2)][(v_1 + 3) $\mapsto$byte(v_2, 1)][(v_1 + 4) $\mapsto$byte(v_2, 0)])
if $\exists \rho \in$ Dom(L) s.t. L = L' $\cup$ {(ρ,R)} and $[v_1, v_1 + 3] \in$ Dom(R.RS)
L else
and byte(n, k) := (n << (8 * (3 - k))) >> 24 .

R11 $\text{(VEnv, L, storecToU region(}\rho\text{), } v_2\text{, } v_1\text{)} \longrightarrow_{stmt} \text{(VEnv, update(L, } v_1, v_2\text{) }\}\text{], } \epsilon\text{)}$

R12 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, poolfree(E, E}_2\text{)} \longrightarrow_{stmt} \text{(VEnv', L', poolfree(E', E}_2\text{)}}$

R13 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, poolfree(}v\text{, E)} \longrightarrow_{stmt} \text{(VEnv', L', poolfree(v, E')}}$

R14 $\text{(VEnv, L} \cup \{ (\rho, \{F; RS\}) \}\text{, poolfree(region(}\rho\text{), v))} \longrightarrow_{stmt} \text{(VEnv, L} \cup \{ (\rho, \{vF; RS\}) \}\text{, } \epsilon\text{)}$ $\boxed{v! = Uninit}$

R15 $\text{(VEnv, L, poolinit(}\rho\text{,}\tau\text{)}x\{ \text{ S } \}\text{)} \longrightarrow_{stmt} \text{(VEnv} \cup \{(x\text{,region(}\rho\text{))}\}\text{, L} \cup \{(\rho, \{\phi; \phi\})\}\text{, pool\{S\}pop(}\rho\text{))}$ if ($\rho \notin$ Dom(L)).

R16 $\dfrac{\text{(VEnv, L, S)} \longrightarrow_{stmt} \text{(VEnv', L', S')}}{\text{(VEnv, L, pool\{S\}pop(}\rho\text{))} \longrightarrow_{stmt} \text{(VEnv', L', pool\{}S'\text{\}pop(}\rho\text{))}}$

R17 $\text{(VEnv} \cup \{(x\text{,region(}\rho\text{))}\}\text{ , L} \cup \{(\rho\text{,R)}\}\text{ , pool\{}\epsilon\text{\}pop(}\rho\text{))} \longrightarrow_{stmt} \text{(VEnv, L, } \epsilon\text{)}$
Note that H the set of addresses in the system heap and not used by the program gets updated by H $\cup$ Dom(R.RS)

Figure 9. Operational semantic rules for statements

$||\texttt{int}||_{(VEnv,L)}$:= Int_{32}
$||\tau * \rho||_{(VEnv,L)}$:= $\{Uninit\} \cup$ Dom(L[ρ].RS)
$||\text{handle}(\rho, \tau)||_{(VEnv,L)}$:= { region(ρ)}
$||\text{Unknown}||_{(VEnv,L)}$:= Int_8
$||\texttt{char}||_{(VEnv,L)}$:= Int_8

Intuitively for a well-formed type τ, $||\tau||_{(VEnv,L)}$ represents the set of values that a variable (or object) of that type can hold under that context and environment. For example, for a variable of type $\tau * rho$, the set of values it can hold are either $Uninit$ or addresses of objects in region ρ, which is Dom(L[ρ].RS)).

Let $\vdash_{env}$ denote the judgment for a well formed environment. We defined an environment (VEnv, L) to be well formed under a typing context C (denoted by $C \vdash_{env}$ (VEnv, L)) if and only if the following invariants hold.

Inv1 Dom(Γ) = Dom(VEnv)

All variables in the typing environment are present in the variable environments and vice versa.

Inv2 Dom(Δ) = Dom(L)

All region names in the region type environment are already present in the domain of region maps and vice versa.

Inv3 $\forall x \in$ Dom(VEnv), if $C \vdash x : \tau$ then VEnv[x] $\in ||\tau||_{(VEnv,L)}$

If a variable has type τ, then it must contain only valid values of type τ. In particular, a pointer variable with region attribute ρ, must always point to an object in that region or it has the value $Uninit$

Inv4 $\forall \rho \in$ Dom(L), if $C \vdash \rho : \tau$ then $\forall v \in$ Dom(L[ρ].RS), L[ρ].RS[v] $\in ||\tau||_{(VEnv,L)}$.

If region ρ is associated with type τ then each memory location in the region store will only contain values of the correct type.

Inv5 $\forall \rho \in$ Dom(L), L[ρ].F $\subseteq$ Dom(L[ρ].RS)

This invariant states that the memory addresses in the free list are a subset of the addresses of the region

Inv6 $\forall \rho_1 \rho_2 \in$ Dom(L), if $\rho_1 \neq \rho_2$ then Dom((L[ρ_1]).RS) $\cap$ Dom((L[ρ_2]).RS) = ϕ and $\forall \rho \in$ Dom(L), Dom(L[ρ].RS) $\cap$ H = ϕ.

A memory address cannot be part of two live regions. Also a memory address cannot be a part of system heap (i.e., unused by a program) and also a part of live region.

Now assume that a run-time check failure leads to the Error state in the operational semantics. We can now prove the following soundness theorem:

R18 (VEnv $\cup \{(x, v)\}$, L, x) $\longrightarrow_{expr}$ (VEnv $\cup \{(x, v)\}$, L, v)

R19 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, E op E}_2\text{)} \longrightarrow_{expr} \text{(VEnv', L', E' op E}_2\text{)}}$

R20 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, } v \text{ op E)} \longrightarrow_{expr} \text{(VEnv', L', } v \text{ op E')}}$

R21 (VEnv, L, m op n) $\longrightarrow_{expr}$ (VEnv, L, m op$_{Int}$ n))

R22 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, load/loadc E)} \longrightarrow_{expr} \text{(VEnv', L', load/loadc E')}}$

R23 (VEnv, L, load/loadc v_1) $\longrightarrow_{expr}$ (VEnv, L, getvalue(L, v_1)) $\boxed{(v_1)! = Uninit}$

where getvalue(L, v_1) := L[ρ].RS[v_1] if $\exists \rho \in$ Dom(L) s.t. $v_1 \in$ L[ρ].Dom(RS)
$Uninit$ else

R24 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, loadFromU/loadcFromU E, E}_2\text{)} \longrightarrow_{expr} \text{(VEnv', L', loadFromU/loadcFromU E', E}_2\text{)}}$

R25 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, loadFromU/loadcFromU } v_1\text{, E)} \longrightarrow_{expr} \text{(VEnv', L', loadFromU/loadcFromU } v_1\text{, E')}}$

R26 (VEnv, L, loadFromU region(ρ), v_1) $\longrightarrow_{expr}$ (VEnv, L, getvalue(L, v_1, 4) }]) $\boxed{(v_1, v_1 + 3] \in \text{Dom(L[}\rho\text{].RS)}}$
where
getvalue(L, v_1, 4) := combine(L[ρ].(RS[v_1]),L[ρ].(RS[$v_1 + 1$]),L[ρ].(RS[$v_1 + 2$]),L[ρ].(RS[$v_1 + 3$]))
if $\exists \rho \in$ Dom(L) s.t. $[v_1, v_1 + 3] \in$ L[ρ].Dom(RS)
$Uninit$ else
and combine(b1, b2, b3, b4) := (b1 << 24) || (b2 << 16) || (b3 << 8) || (b4).

R27 (VEnv, L, loadcFromU region(ρ), v_1) $\longrightarrow_{expr}$ (VEnv, L, getvalue(L, v_1) }])

R28 (VEnv, L, cast E to τ) $\longrightarrow_{expr}$ (VEnv, L, E)

R29 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, castint2ptr E, E}_2 \text{ to } \tau\text{)} \longrightarrow_{expr} \text{(VEnv', L', castint2ptr E', E}_2 \text{ to } \tau\text{)}}$

R30 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, castint2ptr } v\text{, E to } \tau\text{)} \longrightarrow_{expr} \text{(VEnv', L', castint2ptr } v\text{, E' to } \tau\text{)}}$

R31 (VEnv, L, castint2ptr (region(ρ), v to τ) $\longrightarrow_{expr}$ (VEnv, L, v) $\boxed{v \in \text{Dom(L[}\rho\text{].RS)}}$

R32 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, poolalloc(E, E}_2\text{))} \longrightarrow_{expr} \text{(VEnv', L', poolalloc(E', E}_2\text{))}}$

R33 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, poolalloc(}v\text{, E))} \longrightarrow_{expr} \text{(VEnv', L', poolalloc(}v\text{, E'))}}$

R34 (VEnv, L $\cup$ { (ρ, {a F; RS}) }, poolalloc(region(ρ), 1)) $\longrightarrow_{expr}$ (VEnv, L $\cup$ { (ρ, {F; RS}) }, a)

R35 (VEnv, L $\cup$ { (ρ, {ϕ; RS}) }, poolalloc(region(ρ), 1)) $\longrightarrow_{expr}$ (VEnv, L[$\rho \mapsto$ {ϕ; $RS[a \mapsto Uninit]$}], a)
where a is a new address obtained from system allocator, i.e. a $\in$ H. H becomes H - { a }.

R36 (VEnv, L $\cup$ { (ρ, {F; RS}) }, poolalloc(region(ρ), m)) $\longrightarrow_{expr}$ (VEnv, Initialize(L $\cup$ { (ρ, {F; RS}) }, $Uninit$, a,m), a) if (m != 1)
where a is a new address for the array obtained from system allocator and Initialize initializes each element of the array with $Uninit$. H becomes H - { a , a + 1, ... , a + m-1 }

R37 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, (E, \&(E}_1\text{)[E}_2\text{]))} \longrightarrow_{expr} \text{(VEnv', L', E', \&(E}_1\text{)[E}_2\text{']))}}$

R38 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, (}v\text{, \&(E)[E}_2\text{]))} \longrightarrow_{expr} \text{(VEnv', L', } v\text{, \&(E')[E}_2\text{']))}}$

R39 $\dfrac{\text{(VEnv, L, E)} \longrightarrow_{expr} \text{(VEnv', L', E')}}{\text{(VEnv, L, (}v\text{, \&(}v_1\text{)[E]))} \longrightarrow_{expr} \text{(VEnv', L', (}v\text{, \&(}v_1\text{)[E']))}}$

R40 (VEnv, L, (region(ρ), $\&v1[v2]$)) $\longrightarrow_{expr}$ (VEnv, L, $v1 + v2 * \texttt{sizeof}(\tau)$) $\boxed{(v1 + v2 * \texttt{sizeof}(\tau)) \in \text{Dom(L[}\rho\text{].RS)}}$
where τ is the "static" type of the individual element of the array, available from the declaration.
Note that sizeof(τ) is a compile time constant.

Figure 10. Operational semantic rules for expressions

THEOREM 1. *If* $\Gamma \vdash S$ *and* $\Gamma \vdash_{env}$*(VEnv, L) then either (VEnv, L, S)* $\longrightarrow^*_{stmt}$*Error or (VEnv, L, S)* $\longrightarrow^*_{stmt}$ *(VEnv', L', ϵ) and* $C \vdash_{env}$ *(VEnv', L').*

Proof: The proof for this theorem is by induction on the structure of typing derivations (In Section 11 of the technical report [13]).

The soundness result gives us the following invariant – "For a well typed program containing pointer variable p whose declared type is $\tau * \rho$, in every execution state the value of p is guaranteed to be a pointer to an object in the region ρ". This holds even in the presence of undetected memory errors like dangling pointer dereferences and array bound violations, and thus it guarantees correctness of the aliasing information induced by our type system.

3.6 Weaknesses

The key weakness of our system is that it permits dangling pointer errors and array bounds violations to go undetected (but confined within a pool). As explained in Section 1, the only current solution to the former (for obtaining soundness guarantees) is via the use of automatic memory management. For the latter, although we could implement more precise array bounds checks, the best current solution for precise bounds checking imposes more overhead than our approach.

A second issue is that in some cases, our system might require more memory than the original C program (since we cannot free memory to the system until a region goes out of scope). In our previous work, we have evaluated the increase in the context of programs with no type casts and found that the increase is minimal in practice [15]. We believe this issue is unlikely to be significant in practice because we allow reuse within regions (which we believe is quite common for data structures that shrink and grow).

Finally, if the pointer analysis cannot infer an allocation site and consequently a region, for a pointer (e.g. if the address is "manufactured" or read off the disk), we simply insert an `abort`

Region var			ρ
PointerType	pt	:=	$\tau * \rho \mid \tau * (\rho, n)$
Function Type	ft	:=	$\tau \longrightarrow \tau$
Function Sets	fs	:=	f , fs $\mid \epsilon$
FuncPtrType	fpt	:=	ft * fs
StructType	st	:=	struct { $\text{Fld}_1 : \tau_i$, ..., $\text{Fld}_n : \tau_n$ } $\mid \forall \rho$.st $\mid \tau$<ρ>
Type	τ	:=	int | pt | st | ft | fpt | $Unknown$
Expressions	E	:=	$\text{E}_{prev} \mid \&(x\text{->}\text{Fld}_i) \mid \&f$
Defi nitions	d	:=	FSET fs = f, fs

Figure 11. Syntactic extensions for representing structs, functions, function pointers. E_{prev} is expressions, E, of the core language, shown in Figure 1

before every use of such a pointer. This could reject a legal C program (other systems like CCured share the same weakness).

4. Extensions for full C

Several constructs of C were omitted in the previous section to explain our core ideas. Our implementation handles the full language. In this section, we briefly discuss how we handle the remaining constructs including function calls, function pointers and support for region polymorphic functions (we omit control-flow as it does not require any checks and is straightforward to add). The accompanying technical report [13] contains more detailed discussion.

Some of the ideas for implementing region polymorphism in functions and structs are directly borrowed from Cyclone [20]. However, it is worth noting that our universal types are quantified only over region type variables (and not arbitrary type variables). This is sufficient for our goal of trying to retrofit polymorphic region types for otherwise non-polymorphic C code.

Structure types: Structures types are like in C and the syntax for structures is shown in Figure 11. A pointer can point into a structure at an offset $n \geq 0$ and we use $\tau * (\rho, n)$ to denote the type of such a pointer (n is a compile type constant). Given this, the only extra safety implications of structure types are that (a) the `poolcheck` must use the offset o in checking alignment, and (b) structure indexing operations for pointers to TU regions need a poolcheck (similar to array indexing). A notational issue is that it is convenient to include polymorphic type constructors (similar to those used in Cyclone [20]) because a struct type with a pointer field can be used in different places with the field pointing to distinct sets of objects (e.g., when two distinct linked lists are created with the same list node type). The syntax was shown in Figure 11. For example, the polymorphic type `struct S<rho> { Field0 : int, Field1 : int * rho }` can be instantiated with a region type variable to get a new type pointing to a particular points-to set.

Region-polymorphism for functions: Like Cyclone [20], we support region polymorphic functions, parameterized via region names. Region polymorphism is necessary because it is impractical to duplicate function definitions for each context in which they are used. Automatic Pool Allocation already infers this region polymorphism automatically for C programs based on points-to analysis [31]. We leverage that work and only have to type-check that the inferred polymorphism and instantiation are correct.

Function pointers: We represent the call graph in the input type system by adding a function set attribute (called fs in Figure 11) to each function pointer type, making explicit the set of possible targets for that function pointer. The function set attribute can be initialized using the FSET definition. For example, the definition `FSET fs = func1, func2, func3` followed by a use `(int -> int)*fs fptr` denotes a function pointer `fptr` whose targets are the functions `func1, func2, func3`. Before an indirect call, we check at run-time if the function pointer actually points to one of the functions in its `FSET` attribute. A number of these run-time checks are unnecessary and can be eliminated using simple static typing rules. Essentially function pointers that are read from a TU pool (via casts from int to function pointers), and function pointers whose targets are more precise than the one used by the Automatic Pool Allocation, will continue to require a run-time check.

Global and stack allocations: We make memory allocation for both global and stack variables explicit using operations `alloca` and `galloc` (the latter takes an optional initializer). These eliminate the need for the & operator for taking the address of variables. Note that a global or stack object may not have a valid pool handle if no heap object is aliased to it. To ensure that valid pool handles are created for these objects, we pretend that they are allocated using `malloc` at program entry and function entry respectively, and infer the life times of regions using Automatic Pool Allocation. Globals are still allocated in the global area just like the original program. We simply register the valid range of global addresses with the corresponding pool handle if any run-time checks are ever needed for that global pool. Stack objects whose region is created within the same function (i.e., does not escape to a parent) are allocated on the stack, like the original program. Otherwise, we allocate them using `poolalloc` at function entry and free them at function returns. In practice we found that most stack allocations in the original program do not escape and can actually be allocated on the stack.

Compatibility with external libraries: So far we have assumed that we have the source for the complete program including all external libraries. In practice, we have to deal with cases where the sources for some external libraries may not available or it may not be feasible to analyze them. Here, we explain how we handle these external library calls.

Our approach can work correctly (but slightly inefficiently) for most library calls. Our pointer analysis marks any points-to graph node reachable from an external function as "incomplete" (by omitting a **C** (Complete) flag) We can treat pools corresponding to such points-to graph nodes as **TU** pools. Our typing rules then ensure that we can only load/store an `int` or `char` from/into such pools. All pointers read from such memory have to go through a run-time check, because of **R31**, thus ensuring soundness. However, this means we may conservatively perform more checks than necessary.

One case that deserves a special mention here is that of pointers to memory allocated within external library and returned to the program. The pool corresponding to such pointer may not exist (will be null in our implementation) or even if it does exist, because of node merging during the pointer analysis, such a pool does not actually contain the target object of the pointer. This will lead to a run-time failure if the program ever executes a run-time check on pointers to such object. To partially solve this problem, we intercept all calls to malloc/free from the libraries. We store all the allocations in a global hash table, and do a run-time check in the global hash table when a pool is null or if a normal run-time poolcheck fails. However, we can do this only if the memory returned by an external library call is dynamically allocated heap memory. If the library call returns stack allocated memory, or memory in the static region, which we didn't encounter so far in our experiments, we abort the program.

Another problem we encounter is that of call back functions. If an internal function may be called from external code, we must ensure that the external code calls the original function, not the pool-allocated version. This ensures backwards-compatibility but at the cost of soundness. In most cases, we can directly transform the program to pass in the original function and not the pool-allocated version: this change can be made at compile-time if it

passes the function name but may have to be done at run-time if it passes the function pointer in a scalar variable. In the general case (which we have not encountered so far), the function pointer may be embedded inside another data structure. Even for most such functions, the compiler can automatically generate a "`varargs`" wrapper designed to distinguish transformed internal calls from external calls. When this is not possible, we must leave the callback function (and all internal calls to it), completely unmodified.

The third problem is that of incorrect usage of library calls leading to undetected/unmasked memory errors in the unchecked external code. Though we automatically check preconditions for some of the standard C library calls before their invocation, the general solution again involves analyzing the source of the libraries.

5. Implementation

Our compiler system, SAFECode (*S*tatic *A*nalysis *F*or safe *E*xecution of *Code*), is implemented using the LLVM compiler infrastructure [30]. In principle, SAFECode supports any source language translated into the LLVM IR, but our experience has been with C.

5.1 Type inference and type checking

Conceptually, analysis validation in SAFECode consists of a non-standard type-inference step using Automatic Pool Allocation, followed by a standard type checking step using our pool-based type system defined earlier, and insertion of the necessary run-time checks described in Section 3.4.

The "type inference" phase of SAFECode takes the input program and the points-to graph as defined in section 2.1 and transforms the program to add the region type attributes and region parameters of our extended type system. Because our type rules include the region types, region lifetimes, and lexical scoping of region parameters, our type checker effectively ensures the correctness of the region inference.

Our current implementation does more run-time checks than those outlined in the operational semantics; we do poolchecks before all uses of a pointer pointing to TU pool. These checks subsume checks of casts from int to pointer to TU pool (**R31**), checks on indexing of pointers to TU pools (**R40**) but add unnecessary checks before uses of pointers to TU pools read from TK pools. We are refining our implementation to eliminate unnecessary checks.

5.2 The SAFECode runtime system

The pool allocation runtime library requires some significant changes to support the safety guarantees and run-time checks required in this work. The key new aspects of the run-time are briefly described here. A more complete description is available in the technical report [13].

A pool in our implementation is organized as a linked list of (large) blocks. The pool handle (or the pool descriptor) stores the header to this list. If there is insufficient space for a new allocation, the pool requests more blocks from the underlying system heap using malloc. An allocation request is satisfied by returning a free chunk within one block (or spanning multiple blocks if needed). One key change in the pool implementation is that heap metadata (such as the object header describing the size of an allocated object and the free list) cannot be interleaved with live objects in a pool since our approach allows some memory errors to overwrite arbitrary data within a pool. Allowing the metadata to be corrupted would potentially lead to arbitrary safety violations. We maintain metadata for the free list at the start of each free block and ensure (as part of the poolchecks below) that this data cannot be corrupted. To record the size of an allocated object so that it can be found efficiently, we take advantage of type homogeneity (which we have empirically found is available for most pools even in C programs, as explained in Section 2.1). We use a bit vector (with one bit per data element of the pool type) to track the start of each allocated object (or the start of a free chunk immediately after an allocated object). Because searching this bit vector would be very inefficient for large arrays, we allocate each large array in a (contiguous) set of new blocks and perform a `poolfree` for the array simply by freeing all the blocks.

By far the most important operations (in terms of performance impact) are the pool bounds checks (`poolcheck`), which are used either during the array indexing or during cast operations. To make the check efficient, we organize heap memory in the pool as blocks of size 2^k bytes for some fixed k, and record the starting address of all blocks in a hash set. For `poolcheck`(ph, A, o): we check $A \ \& \ \tilde{}(2^k - 1)$ (mask off last k bits) against the hash set of `ph`. Alignment check is straightforward since each block contains objects of the same type. To exploit the high spatial locality of array references, we use a two-element cache to remember the block address of the two last successful hash lookups.

When a reserved address range is available (e.g., the high GB within a 4GB address space for processes on Linux), we set `Uninit` to the base of this range so that the `Uninit` check is performed "for free" by the memory management hardware. This technique is unusable for kernel modules and also for references that may access a structure type with size greater than the reserved range (which is extremely rare). In such cases, we have to retain explicit software checks at run-time.

6. Sound Static Analyses Enabled By SAFECode

The guarantees provided by our system can be used to write sound static analyses based on the points-to graph, call graph, and type information. In this section we first show that a static array bounds checking technique developed in our previous work, which relies on a call graph, can now be used soundly for non-type-safe programs in our environment. We also illustrate how our soundness guarantees about alias analysis can benefit other static analysis tools, using an existing software verification tool as an example.

6.1 Static array bounds checking in SAFECode

We can use an interprocedural array bounds checking algorithm that we developed previously (for a type-safe subset of C) to eliminate some runtime array bounds checks. The algorithm uses the call graph but not points-to-graph because it does not track values through loads/stores. It propagates affine constraints on integer variables from callers to callees (for incoming integer arguments and global scalars) and from callees to callers (for integer return values and global scalars). We then perform a symbolic bounds check for each index expression using integer programming (our compiler uses the Omega Library from Maryland [28]). We retain the run-time checks for all the array references that could not be proved to be safe using our static analysis. Since SAFECode semantics guarantee the correctness of the call graph, this optimization is safe (just like it would be safe for a type-safe language). To our knowledge, SAFECode is the first system for ordinary C programs (including explicit memory deallocation) where such an optimization can be performed safely.

6.2 Static analyses in ESP

As a final example, we briefly describe one software validation tool, ESP [10], that relies on alias analysis to give guarantees about programs and could benefit from the guarantees provided by our system. Other software validation tools, e.g., BLAST [24], could make use of our guarantees in a similar fashion.

ESP relies on *value flow analysis* [16], a static analysis used to identify the set of pointer expressions that refer to the memory locations holding a certain value of interest, such as a lock. These

```
void KernelEntryPoint(int **o) {
   int **q, *r;
1: r = malloc(....);
2: ... //some computation using r
3: free(r);
   if (o != NULL)
4:      q = o;
   else {
5:      q = malloc(..);
6:      *q = ... /* *q initialized with some safe value */
   }
7: *r = ... /* dangling pointer error, can overwrite *q */
8: if (o != NULL)
       Probe(o);   /* checks that *o is a valid pointer */
9: **q = data1; /* Dereference arbitrary pointer */
}
```

Figure 12. Example – Value flow analysis with memory errors

sets are called *value alias sets* and computed by a data-flow analysis (*value flow simulation*). The dataflow transfer functions disambiguate memory references using flow-insensitive, unification-based context sensitive may-alias analysis. This approach has been used to verify various properties in software, e.g., the Probe security property [16], which requires that any pointer passed into the kernel from user space is checked ("probed") before being dereferenced by the kernel.

Consider applying ESP to verify the code fragment in Figure 12, which is a buggy version of the kernel code fragment used in [16]. In the function, `KernelEntryPoint`, the pointer `o` is passed in from a user routine and its target needs to be *probed* before being dereferenced by the kernel. Because of line 4, ESP tracks `q` and `o` as value aliases if `o != NULL`. The newly allocated memory when `o == NULL` is initialized to be *safe*. Line 7 contains a memory error (a dangling pointer dereference). Since the system memory allocation could allocate previously freed memory of r for the allocation of `q`, this dangling pointer dereference could actually overwrite `*q`. This violates the results of the May-alias analysis that `q` and `r` are not aliased to each other. In line 8, the target of pointer `o` is probed. ESP thus transitions both the value aliases, `o` and `q`, to the *safe* state. Dereferencing `*q` is hence detected to be safe by ESP. However, in reality, `*q` could now point to any location in memory and can be dereferenced by the program, violating the Probe security property. Enforcing the assumed aliasing properties is essential for the soundness of the tool.

In our system, the same example would allocate `q` and `r` in two different pools as they are not aliased. This ensures that the dangling pointer error in `r` does not trample the memory of `q`. Thus, we ensure the validity of a critical aliasing property, without actually detecting the error itself.

The above is an example of a flow-sensitive program analysis that uses an external flow-insensitive alias analysis and can be easily made sound using our approach. For a general flow-sensitive analysis that reasons about loads/stores, we must modify the semantics of malloc (and free) in the analysis so that the address returned by malloc may be "aliased" to any previously freed objects in the same alias set. This is a straightforward (and local) change within the implementation of a dataflow analysis.

7. Results

We present an experimental evaluation of SAFECode for several ordinary C programs and a few operating system daemons. These experiments have three goals:

- To measure the net overhead and different components of overhead incurred by our run-time checks;
- To evaluate the benefit of using sound static analyses enabled by SAFECode to eliminate various kinds of runtime checks.
- To compare the overhead of our approach to that of CCured.

7.1 Run-time Overheads

We evaluated our system using 9 programs from the Olden suite of benchmarks [8], 3 programs from PtrDist, and four system codes – bsd-fingerd-0.17, ftpd-BSD-0.3.2, ghttpd-1.4, and netkit-telnet-0.17 daemon. The benchmarks and their characteristics are listed in Table 1. We compiled each program to the LLVM compiler IR, perform our analyses and transformations, then compile LLVM back to C and compile the resulting code using GCC 3.4.2 at -O3 level of optimization. For the benchmarks we used a large problem size to obtain reliable measurements. For `ftpd` and `fingerd`, we ran the server and the client on the same machine to avoid network overhead, and measured their response times for client requests. We successfully applied SAFECode to `netkit-telnetd` but this is an interactive program and we did not notice any perceptible difference in the response times. We do not report detailed timings for this code here.

The "native" and "LLVM (base)" columns in the table represents execution times when compiled directly with GCC -O3 and with the base LLVM compiler using the LLVM C back-end followed by GCC -O3. Using LLVM (base) times as our baseline allows us to isolate the overheads added by SAFECode. The "PA", "PA + non-array checks", and "SAFECode" columns show the execution times with just pool allocation, SAFECode without array indexing checks, and SAFECode with all the run-time checks respectively.

The column "SAFECode/PA" (the ratio of SAFECode time to pool allocation time) shows that the run-time checks added by SAFECode have a relatively small impact on performance (over and above pool allocation): less than 10% in all cases except `ks` and `yacr2`, which have 11% and 18% overhead. The latter two overheads are entirely due to pool checks for array references, as seen by comparing the "PA+non-array checks" vs. the "SAFECode" columns.

Comparing the columns "SAFECode/LLVM" (ratio of SAFECode time to LLVM base time) with "SAFECode/PA," we see that the pool allocation transformation has a significantly bigger impact on performance than the run-time checks. Four of the programs show significant slowdowns due to PA: `em3d`, `anagram`, `ks` and `yacr2`. We believe that these slowdowns are because our modified pool run-time library has not been tuned at all. We currently use an inefficient bit-vector implementation of free lists. A more recent version of the pool runtime library used in [31] shows no slowdown for these four programs. We aim to merge our extensions with this version in the near future.

In case of `perimeter`, we discovered that LLVM uses "loop invariant code motion" to remove an expensive computation out of a timing loop, thus dramatically speeding up its performance compared to gcc. This suggests that SAFECode/LLVM ratio is the only meaningful way to isolate the overheads of SAFECode approach. `voronoi` benchmark fails at run-time because our pointer analysis is currently unable to track the region for a pointer that is cast from an int and it is treated as a "manufactured" pointer.

7.2 CCured comparison

The last column in Table 1 compares the overhead of SAFECode with that of CCured, for the Olden benchmarks rewritten by the CCured team. We have not tried to compare our results on other system codes as it involved significant porting effort in writing the CCured wrappers. In all these programs SAFECode has significantly less overhead than CCured, even though SAFECode's pool checks are more expensive than the run-time checks inserted by

Benchmark	Lines	Execution times (secs)						Slowdown ratios			
	of code	native	LLVM (base)	PA	PA + non array checks	SAFECode	CCured	SAFECode /LLVM	SAFECode /PA	SAFECode /native	CCured /native
Olden											
bh	2053	1.449	1.357	1.338	1.361	1.403	1.923	**1.03**	1.05	0.97	**1.31**
bisort	707	11.740	11.530	11.531	11.531	11.531	11.358	**1.00**	1.02	0.98	**0.97**
em3d	557	13.960	11.29	14.245	14.245	14.248	20.812	**1.27**	1.00	1.02	**1.49**
health	725	1.909	1.936	1.296	1.296	1.299	1.710	**0.67**	1.00	0.68	**.90**
mst	617	11.259	12.920	12.837	12.837	12.96	16.956	**1.00**	1.01	1.15	**1.51**
perimeter	395	2.033	0.048	0.051	.051	0.051	2.544	**1.04**	1.00	.025	**1.25**
power	763	1.253	0.887	0.934	0.934	0.918	1.408	**1.03**	0.98	0.73	**1.12**
treeadd	385	5.426	5.457	5.425	5.425	5.425	14.784	**0.99**	1.00	1.00	**2.72**
tsp	561	1.277	1.270	1.250	1.250	1.250	1.578	**0.98**	1.00	0.98	**1.23**
voronoi	111	Rejected because of cast from integer to pointer									
System											
fingerd	338	6.410	6.555	6.617	6.617	6.753	-	**1.03**	1.02	1.05	-
ftpd	26653	1.210	1.185	1.160	1.160	1.190	-	**1.00**	1.03	0.98	-
ghttpd	837	3.723	3.507	3.761	3.780	3.766	-	**1.07**	0.99	1.00	-
PtrDist											
anagram	647	12.778	16.084	16.915	17.953	19.742		**1.23**	1.05	1.54	-
ks	782	3.554	4.429	4.501	4.501	4.981	-	**1.12**	1.11	1.40	-
yacr2	3982	3.795	3.991	4.398	4.398	5.204	-	**1.30**	1.18	1.37	-

Table 1. Benchmarks (telnetd in text) - Runtime Overheads

CCured. The lower overhead can be attributed to the broad range of static analysis techniques employed by SAFECode for eliminating garbage collection (GC) overhead, stack safety checks, and many array bounds checks, and the run-time techniques that eliminate null pointer checks and metadata maintenance overhead. Note, however, that several of our static and run-time techniques for reducing overhead (except GC overhead) could be used with CCured as well. We believe that for end-users, any differences in the overheads of the systems is likely to be less important than the choice between automatic and explicit memory management.

7.3 Effectiveness of Static Analysis

Table 2 shows the effectiveness of our static checks and of segregating memory objects into TK and TU pools. Columns 2 and 3 show the total number of static array accesses and the number that must be checked at run time. The next two columns show the total number of static loads and stores and the number of pointers that need to be checked at run time. The last two columns show the static number of **TU** and **TK** pools. We found that our static array safety checks were successful in eliminating some run-time array bounds checks in most programs. Our static pointer safety techniques eliminate all other run-time checks (checks involving **TU** pools), except in the three programs that have **TU** pools.

Benchmark	Static Counts					
	Total array accesses	Checked array accesses	Total loads / checks	Non-array pointer checks	TU	TK
bh	80	45	708	96	1	3
bisort	2	0	103	0	0	1
em3d	17	14	80	0	0	10
health	3	0	221	0	0	2
mst	4	3	53	0	0	5
perimeter	4	4	233	0	0	1
power	4	4	229	0	0	4
treeadd	2	0	31	0	0	1
tsp	0	0	176	0	0	1
fingerd	13	8	32	11	0	3
ftpd	362	209	1949	285	2	22
telnetd	432	363	1602	0	0	15
anagram	63	47	164	4	1	5
ks	58	52	326	0	0	3
yacr2	302	302	856	0	0	26

Table 2. Benchmarks - Effectiveness of Static Checks

8. Related Work

For weakly typed languages like C and C++, there are broadly two kinds of techniques addressing memory errors: memory safety techniques that try to detect or prevent some or all memory errors, and stronger approaches that provide soundness guarantees. In this section, we discuss each of these different kinds of techniques along with few other related approaches.

8.1 Techniques focusing on detecting memory errors:

Purely static approaches: A number of techniques have been proposed to detect memory errors at the source level. A majority of these techniques target bounds errors [5, 15, 17, 18, 40, 21]. Some of these (including EspX [21], CSSV [17]) completely eliminate buffer over runs by requiring annotations in the source and checking them. While these annotations, when written, are extremely useful to eliminate the bounds errors, we believe that majority of todays software may never be rewritten with annotations. Other static approaches for bounds detection([5, 15, 18, 40]) including the static bounds checker developed in our previous work [15], do not use annotations, but generate many false positives when applied to general programs. We believe these approaches are complementary to our approach here and can be used to prove some array accesses as safe and eliminate those run-time checks. Similarly static techniques that detect other memory errors like pointer dereferences to freed memory (e.g., [22]) also generate false positives and can only be used to optimize away some of the run-time overhead. Thus we deem them as complementary.

Run-time techniques for error detection: There have been a large number of systems for detecting memory access errors by adding run time checks and meta-data [23, 35, 2, 27, 34, 32, 39, 41] (the work by Loginov at el. also detects type errors [32]). Except the Patil work [34], these systems use heuristic techniques that do not detect or eliminate all possible errors, especially dangling

pointer errors which are quite difficult to detect reliably. Therefore, these systems do not provide a sound basis for static analysis techniques. The tool by Patil and Fisher [34] can reliably detect memory reference errors, including dangling pointer errors but at the cost of very high overheads (2x-6x in many programs). Furthermore, even this tool does not prevent type violations on references to legal memory addresses (though it might be extended to do so). Overall, none of these tools provide a sound semantics despite their high-overhead run-time checks.

In other work, we have proposed a new backwards-compatible technique to detect all bounds errors precisely [11]. This extends Jones-Kelley's backwards-compatible bounds checking technique [27] to detect bounds errors but *without* the reported 10x-11x overhead. The technique relies on pools to partition the memory and uses a table for each pool to lookup the source of each pointer arithmetic, thus avoiding the metadata and backwards compatibility concerns. However, because of the precise checks, the overheads are still higher compared to our current approach; a maximum of 69% compared to a maximum of 30% in the current work on the same set of Olden benchmarks.

More recently, we have also investigated a technique that can detect dangling pointer errors [12] instead of masking them. This work makes use of virtual memory checks performed in hardware and pools to detect dangling pointer errors. However, that work targets server software where the memory allocation/deallocation frequency is less and is not applicable to other classes of software with frequent allocations and deallocations.

8.2 Techniques providing a strong guarantee

Two systems, CCured [33] and Cyclone [20], both enable type-safe execution of C or modified C programs, which enables sound analysis of these programs. CCured ensures type-safe execution for standard C programs, with some source changes required for compatibility with external libraries. It uses a conservative garbage collector instead of explicit deallocation of heap memory. Compared with our approach, the major advantage of CCured is that it guarantees the absence of dangling pointer references and also performs exact bounds checks on all memory references. In contrast, a key contribution of our work has been to enable sound analysis while still retaining explicit memory management. A second difference is that CCured introduces significant metadata for runtime checks. This metadata is the primary cause of the porting effort required for using CCured on C programs because it can require wrappers around some library functions. SAFECode uses no metadata on individual pointer values and provides better backwards-compatibility than CCured.

There are also minor technical differences between the systems. Our classification of memory into type-consistent and $Unknown$ is analogous to the WILD and non-WILD types of CCured, except that we use a pointer analysis to infer the types of memory objects. We allow $Unknown$ memory to point to type consistent memory by performing a run-time check as explained in Section 3.1. CCured uses physical subtyping and RTTI to eliminate some run time overhead on pointer casts. Our type inference supports limited forms of physical subtyping (only for upcasts and casts from void* to other pointer types) but we plan to investigate a more sophisticated version in the future.

Cyclone [20, 25] uses a region-based type system to enforce strict type safety, and consequently enforces alias analysis, for a variant of C. Unlike SAFECode and CCured, Cyclone disallows non-type-safe memory accesses (e.g., operations that would produce the equivalent of $Unknown$ type or WILD pointers). Cyclone and other region-based languages [6, 19, 7, 9, 37]) have two disadvantages relative to our work: (a) they can require significant programmer annotations to identify regions; and (b) either they provide no mechanism to free or reuse memory within a region (e.g., RT-Java) or they allow deallocation of memory within a region only in special cases (e.g., uniqueness annotations to Cyclone [25] or reset region in ML kit for regions [37]). In all the above systems, data structures that must shrink and grow (with non-nested object lifetimes) can be put in regions only when they use a restricted form of aliasing. Often they have to be allocated on the garbage collected heap. In contrast, we infer the pool partitioning automatically with no annotations, and we permit explicit deallocation of individual data items within regions without aliasing restrictions or extra annotations.

8.3 Other related work

Reaps [4] is another region based system that is related to our work. Reaps are regions with efficient deallocation of individual objects within a region. However Reaps is a performance enhancement approach and does not provide any soundness guarantees.

DieHard [3] is another related approach that tries to achieve probabilistic error tolerance using randomization of the allocations/deallocations in the heap and replication of the program. DieHard uses a heap, which is M times the maximum live size, and relies on randomization to avoid the bounds errors and dangling free errors. The execution time overheads of DieHard in the non-replicated version are less than 8%. However, the possible several fold increase in memory consumption could make it unattractive for system software. Moreover, running several copies of a program (replication) may be suitable only in cases where spare processors are readily available. Finally, DieHard ignores errors on stack memory. In contrast, we provide strong guarantees on the correctness of our semantics and do it for stack, heap, and global memory, and incur minimal increases in memory consumption.

9. Concluding Discussion

This paper has described an approach to provide a semantic foundation (a points-to graph, call graph, and type information) for building sound static analyses for nearly arbitrary C programs. The approach can be easily added to any C compiler containing a pointer analysis that meets the specified properties (flow-insensitive, unification-based).

The approach also has some other practical strengths: it is fully automatic and requires no modifications to existing C programs; it allocates and frees memory objects at the same points as the original program (minimizing the need to tune memory consumption); and it supports nearly the full generality of the C language, except for "manufactured addresses" (which could also be supported via pragmas or compile-time options) and some casts from int to pointers. Finally, our experiments show that the run-time overheads of our approach are quite small, generally less than a few percent relative to code with pool allocation alone. We believe these overheads are low enough to be used in production code, especially when security is a significant concern.

We believe that our approach represents an interesting and useful low overhead alternative to techniques that focus on complete soundness with no dangling pointers. However, in some application domains stronger guarantees than given by our current approach are required. We believe that the right long term approach is to offer a choice to the end user between our current approach with alternatives that can detect all memory errors or use garbage collection.

Towards this end, we plan to build a framework that includes this work along with our other work that detects bounds errors [11] and dangling pointer errors [12]. We believe that there shouldn't be any major technical challenges in integrating the two techniques with SAFECode since they are also developed using the LLVM compiler system and Automatic Pool Allocation transformation.

References

[1] A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 1995.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 1994.

[3] E. Berger and B. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2006.

[4] E. Berger, B. Zorn, and K. McKinley. Reconsidering custom memory allocation. In *Proc. Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 2002.

[5] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2000.

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

[7] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2003.

[8] M. C. Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, 1996.

[9] W.-N. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2004.

[10] M. Das, S. Lerner, and M. Siegle. Esp: Path-sensitive program verification in polynomial time. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, Jun 2002.

[11] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. 28th Int'l Conf. on Software Engineering (ICSE)*, Shanghai, China, May 2006.

[12] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proc. Int'l Conf. on Dependable Systems and Networks (DSN)*, Philadelphia, USA, June 2006.

[13] D. Dhurjati, S. Kowshik, and V. Adve. Enforcing alias analysis for weakly typed languages. Tech Report UIUCDCS-R-2005-2657, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Oct 2005. See `http://safecode.cs.uiuc.edu/`.

[14] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Conf. on Language, Compiler, and Tool Support for Embedded Systems (LCTES)*, Jun 2003.

[15] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems*, Feb. 2005.

[16] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *Proc. of ACM SIGSOFT international symposium on Software testing and analysis*, 2004.

[17] N. Dor, M. Rodeh, and M. Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *SIGPLAN Conference on Programming Language Design and Implementation*, Sandiego, June 2003.

[18] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security*, New York, NY, USA, 2003.

[19] D. Gay and A. Aiken. Memory management with explicit regions. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 313–323, Montreal, Canada, 1998.

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

[21] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *Proc. 28th Int'l Conf. on Software Engineering (ICSE)*, Shanghai, China, 2006.

[22] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 310–323, New York, NY, USA, 2005. ACM Press.

[23] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX*, 1992.

[24] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239, 2003.

[25] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in Cyclone. In *Proc. of the 4th international symposium on Memory management (ISMM)*, 2004.

[26] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

[27] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[28] 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.

[29] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Comp. Sci. Dept., Univ. of Illinois, Urbana, IL, May 2005.

[30] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, San Jose, Mar 2004.

[31] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun 2005.

[32] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. *Lecture Notes in Computer Science*, 2001.

[33] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Language and Systems*, 27(3):477–526, 2005.

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

[35] J. Seward. Valgrind, an open-source memory debugger for x86-gnu/linux.

[36] B. Steensgaard. Points-to analysis in almost linear time. In *ACM symposium on Principles of programming languages (POPL)*, 1996.

[37] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, P. Sestoft, and P. Bertelsen. Programming with Regions in the ML Kit. Technical Report DIKU-TR-97/12, 1997.

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

[39] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[40] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Softw. Eng. Notes*, 28(5):327–336, 2003.

[41] S. H. Yong and S. Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Foundations of Software Engineering*, 2003.