CCured: Type-Safe Retrofitting of Legacy Software

GEORGE C. NECULA, JEREMY CONDIT, MATTHEW HARREN,
SCOTT McPEAK, and WESTLEY WEIMER
University of California, Berkeley

This article describes CCured, a program transformation system that adds type safety guarantees to existing C programs. CCured attempts to verify statically that memory errors cannot occur, and it inserts run-time checks where static verification is insufficient.

CCured extends C's type system by separating pointer types according to their usage, and it uses a surprisingly simple type inference algorithm that is able to infer the appropriate pointer kinds for existing C programs. CCured uses physical subtyping to recognize and verify a large number of type casts at compile time. Additional type casts are verified using run-time type information. CCured uses two instrumentation schemes, one that is optimized for performance and one in which metadata is stored in a separate data structure whose shape mirrors that of the original user data. This latter scheme allows instrumented programs to invoke external functions directly on the program's data without the use of a wrapper function.

We have used CCured on real-world security-critical network daemons to produce instrumented versions without memory-safety vulnerabilities, and we have found several bugs in these programs. The instrumented code is efficient enough to be used in day-to-day operations.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Languages, Reliability, Security, Verification

Additional Key Words and Phrases: Memory safety, pointer qualifier, subtyping, libraries

1. INTRODUCTION

CCured is a program transformation system that adds memory safety guarantees to C programs. It first attempts to find a simple proof of memory safety for

This research was supported in part by the National Science Foundation Grants CCR-9875171, CCR-0085949, CCR-0081588, CCR-0234689, CCR-0326577, CCR-00225610, and gifts from Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Authors' addresses: Electrical Engineering and Computer Science Dept., 783 Soda Hall, University of California, Berkeley, Berkeley, CA 94720-1776; email: {necula,jcondit,matth,smcpeak, weimer}@cs.berkeley.edu.

the program, essentially by enforcing a strong type system. Then, the portions of the program that do not adhere to the CCured type system are checked for memory safety at run-time.

Since CCured enforces memory safety, which is an implicit requirement of every C program, it is a good debugging aid. We were able to find several bugs in the Spec95 benchmark suite and in network daemons by running the instrumented programs on their own test suites. Memory safety is also beneficial in extensible systems, such as the Apache Web server or an operating system kernel, which support pluggable modules and device drivers. By instrumenting modules with CCured, the failure of an individual component cannot contaminate the system as a whole.

Perhaps the greatest potential impact of CCured is in the domain of security-critical software. Memory safety is an absolute prerequisite for security, and it is the failure of memory safety that is most often to blame for insecurity in deployed software [Wagner et al. 2000]. Further, CCured's relatively modest performance cost makes it plausible for security-critical production systems to use binaries compiled with CCured's run-time checks enabled.

The work described in this article is based on two main premises. First, we believe that, even in programs written in unsafe languages like C, a large part of the program can be verified as type safe at compile time. The remaining part of the program can be instrumented with run-time checks to ensure that the execution is memory safe. The second premise of our work is that, in many applications, some loss of performance due to run-time checks is an acceptable price for type safety, especially when compared to the alternative cost of redesigning the system in a type-safe language.

The main contribution of this article is the CCured type system, a refinement of the C type system with separate pointer kinds for different pointer usage modes. In a typical C program, CCured discovers that most pointers are used safely, requiring just a null check before dereference. We call such pointers `SAFE`. They are as cheap to use as a reference in a type-safe language such as Java. For other pointers, CCured discovers that they are involved in pointer arithmetic and thus require bounds checks before dereference. These pointers are called `SEQ` ("sequence") pointers. Finally, some pointers are involved in type casts that prevent CCured from tracking the type of the referenced data at compile type. For these `WILD` pointers, CCured adds both bounds checking and run-time type checking, in a manner similar to references in a dynamically typed language like Lisp.

In general, the CCured type system is similar to Lisp or Scheme soft typing [Cartwright and Fagan 1991; Wright and Cartwright 1997]. CCured treats C as a dynamically typed language but optimizes away most of the run-time checks and the state needed to make such checks by recognizing most pointers as `SAFE` rather than `SEQ` or `WILD`. Since Scheme and C are very different, we introduce a novel types and a new inference algorithm to achieve this goal.

CCured can type-check and instrument programs annotated with pointer qualifiers. Since the CCured typing rules are very close to those of C, we believe that it would be easy for C programmers to use CCured's type system in newly written programs. However, it is impractical to make pervasive changes in large existing programs. To address this problem, we designed a pointer-kind inference algorithm that performs a simple linear-time whole-program analysis to discover the best pointer qualifier for each pointer in the program. Specifically, our algorithm chooses a set of pointer qualifiers that minimize run-time overhead while still providing memory safety guarantees.

An early version of CCured, including only the pointer qualifiers described above, was already usable on existing C programs [Necula et al. 2002a]. However, we discovered that there were some serious usability problems that made it difficult to apply CCured to system software and to large security-critical network daemons [Condit et al. 2003]. The major problem was due to incompatibilities between the CCured representation of types and the ordinary C representation used by precompiled libraries. One notable example is the multiword representation of the CCured `SEQ` and `WILD` pointers, which contain additional information (*metadata*) necessary for performing the run-time checks. Also, objects referenced by `WILD` pointers must contain tags used to perform CCured's run-time checks, which makes the problem of library compatibility especially challenging in the presence of `WILD` pointers. Even a small number of casts that CCured considers "bad" can result in a large number of `WILD` pointers, because any pointer that is obtained from a `WILD` pointer through assignment or dereference must be `WILD` as well.

We designed a three-point solution to alleviating the incompatibility problem. First, we observed that in the presence of structures and arrays, most type casts could be classified as either *upcasts* (e.g., from a pointer to an object to a pointer to the first subobject) or *downcasts* (in the opposite direction). To avoid treating these casts as bad, and thus to reduce the number of `WILD` pointers, we extended the CCured type system with a *physical subtyping* mechanism for handling the upcasts and with a special kind of pointer that carries *run-time type information* for handling the downcasts. These two mechanisms, described in detail in Section 3, allow CCured to handle object-oriented techniques such as subtyping polymorphism, dynamic dispatch, and checked downcasts, which are surprisingly common in large C programs.

Second, we developed a notation that allows a programmer to specify the conversions and run-time checking operations that must occur at the boundary between the code processed with CCured and precompiled libraries. In Section 6.1, we describe our technique for automatically instantiating user-specified wrappers in many contexts.

Finally, to address the remaining compatibility problems, we devised a new representation for wide pointers in which CCured's metadata is not interleaved with the program data. Rather, the metadata is stored in a separate data structure whose shape mirrors that of the program data itself. This separation incurs a slight performance penalty due to lost locality and to an increase in the overall amount of metadata; thus, the CCured inference algorithm has been extended to limit the use of this new representation to those types where it is required in order to preserve both soundness and compatibility. This mechanism is described in more detail in Section 6.2.

Even with CCured's simple and intuitive type system, we can limit the use of expensive pointer kinds to a relatively small number of pointers. In most programs, static counts indicate that less than 1% of all the pointers in the program are WILD and that less than 10% of all pointers in the program are SEQ.

In addition, we have produced CCured-transformed versions of several popular network servers (ftpd, bind, openssl/sshd, and sendmail). We have verified that CCured prevents known security exploits, and most importantly, we have produced memory-safe versions of these applications that should eliminate any further vulnerabilities due to memory safety violations. We describe our experience using CCured for these applications in Section 8. Perhaps if these instrumented binaries saw wide adoption we might see an end to (or at least a slower pace of) the cycle of vulnerability reports and patches that has become all too common with security-critical infrastructure software.

The rest of this article is organized as follows. We start with a description of the CCured type system and instrumentation in Section 2. In Section 3 we refine this type system with the notion of physical subtyping and pointers with run-time type information. In Section 4 we describe formally the invariants that are maintained by the typing rules and we state the type soundness theorem. In Section 5 we describe informally the handling of additional C programming constructs, and in Section 6 we describe the mechanisms used for ensuring compatibility with existing libraries. Section 7 gives an overview of the work involved in applying CCured to a typical C program. Then, Section 8 shows the performance of CCured-instrumented programs; in addition, we discuss bugs we found and techniques we applied. Finally, we compare CCured with related work in Section 9, and present our conclusions in Section 10.

2. CCURED TYPES AND CHECKS

The CCured type system can be viewed as two universes that coexist soundly. On the one hand, we have statically typed pointers for which we maintain the invariant that the static type of the pointer is an accurate description of the contents of the referenced memory area. On the other hand, we have dynamically typed pointers, for which we cannot rely on the static type; instead, we rely on run-time tags to differentiate pointers from nonpointers.

CCured has different pointer kinds with varying capabilities and costs. Typically, most pointers in C programs are used without casts or pointer arithmetic. We call such pointers SAFE, and they are either null or valid references. Pointers that are involved in pointer arithmetic but are not involved in casts are called SEQ ("sequence") pointers. SEQ pointers carry with them the bounds of the array into which they are supposed to point. The SAFE and SEQ pointers are statically typed in the sense that the type system ensures that their static type is an accurate description of the objects to which they refer.

In order to analyze type casts, we introduce a notion of type compatibility that is described later in this section. A type cast between incompatible types is called a *bad cast*. The pointers involved in a bad cast are considered to be dynamically typed, or WILD. Such pointers have all the capabilities of C pointers; however, the static type of a WILD pointer is not necessarily an accurate description of the contents of the memory to which it refers.

Figure 1 presents the syntax of types and expressions for a simple C-like programming language that serves as the vehicle for formalizing CCured. We

$$\begin{array}{lll}
\text{Atomic types:} & \alpha ::= & \texttt{int} \mid \tau * \texttt{SAFE} \mid \tau * \texttt{SEQ} \mid \tau * \texttt{WILD} \\
\text{Types:} & \tau ::= & \alpha \mid \texttt{struct}\{\ \tau_1 f_1; \ldots \tau_n f_n;\ \} \mid \texttt{void} \mid \tau[n] \\
\text{Expressions:} & e ::= & x \mid n \mid (\tau)e \mid e_1 + e_2 \mid *e \mid *e_1 = e_2 \mid \texttt{malloc}(e)
\end{array}$$

Fig. 1. The syntax of a simple language with pointers, structures, pointer arithmetic, and casts.

$$\begin{array}{lll}
\mathsf{Rep}(\texttt{int}) & = & \texttt{int} \\
\mathsf{Rep}(\texttt{struct}\{\ \ldots\tau_i f_i; \ldots\ \}) & = & \texttt{struct}\{\ \ldots\mathsf{Rep}(\tau_i) f_i; \ldots\ \} \\
\mathsf{Rep}(\texttt{void}) & = & \texttt{void} \\
\mathsf{Rep}(\tau[n]) & = & \mathsf{Rep}(\tau)[n] \\
\mathsf{Rep}(\tau * \texttt{SAFE}) & = & \texttt{struct}\{\ \mathsf{Rep}(\tau) * p;\ \} \\
\mathsf{Rep}(\tau * \texttt{SEQ}) & = & \texttt{struct}\{\ \mathsf{Rep}(\tau) * p, * b, * e;\ \} \\
\mathsf{Rep}(\tau * \texttt{WILD}) & = & \texttt{struct}\{\ \mathsf{Rep}(\tau) * p, * b;\ \}
\end{array}$$

Fig. 2. CCured representations. Given a CCured type τ (with pointer qualifiers), $\mathsf{Rep}(\tau)$ gives its layout and representation.

distinguish the atomic types (scalars and qualified pointers) from the rest of the structured types. The type void is equivalent to an empty struct type. The memory read expression, pointer arithmetic expression, and memory write expression will be the most important expressions from the perspective of memory safety.

We make a number of simplifying assumptions in the following discussion. First, we assume that the machine word size is one byte. Second, we assume that all user variables have atomic types. Third, we model function calls as assignments to the parameters and from the return value. And fourth, since our analysis is flow-insensitive, we omit control-flow details.

We do not consider recursive types here, but they can easily be added to our framework. We also do not consider union or function pointer types, which complicate the implementation in a significant way; we will discuss these issues informally in Section 5.

Finally, this section considers only the three main pointer kinds: SAFE, SEQ, and WILD. Our implementation of CCured uses a number of additional pointer kinds. For example, the FSEQ ("forward sequence") pointer kind is an optimization of SEQ for the common case in which pointer arithmetic only moves a pointer towards higher addresses. Therefore, FSEQ pointers do not need to remember the lower bound of the array. Also, Section 3.2 introduces a pointer kind that carries run-time type information. Since these kinds are inferred from constraints analogous to those for SEQ, we do not discuss them further in this section.

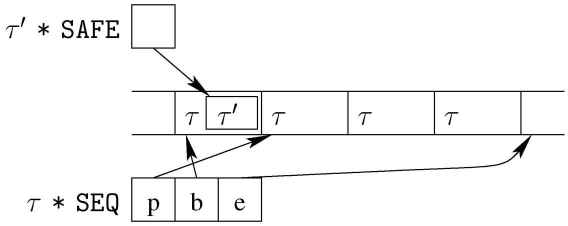
2.1 Representation

In this section, we discuss the informal invariants that are maintained by each kind of CCured pointer. These invariants are then used in the next section to describe the static and run-time checks that CCured performs. The formal discussion of the invariants along with a soundness argument is given in Section 4.

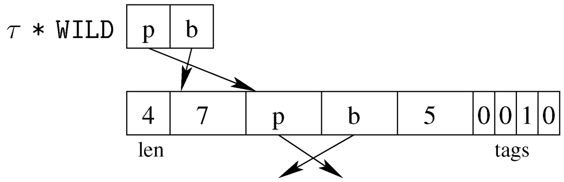
Since CCured's WILD and SEQ pointers must carry bounds and/or type information, they are represented differently from normal C pointers. The representation for a CCured type τ is given by the function $\mathsf{Rep}(\tau)$, defined in Figure 2. Note that each pointer kind is represented by a structure containing pointers labeled p, b, and e. The pointer p in each representation corresponds to the original C pointer; the base pointer b and end pointer e hold CCured metadata and cannot be independently modified by the program. We refer to the pointer, base, and end parts of a CCured pointer x as x.p, x.b, and x.e, respectively.

First, consider the SAFE pointer. A pointer of type τ * SAFE, if not null, is guaranteed to point to a memory area that contains an object whose type is compatible with τ. Since such pointers require no bounds information, a single pointer p suffices for the representation.

Next, we examine the SEQ pointer. The representation of this pointer kind includes the pointer itself plus a base pointer and an end pointer, which contain bounds information. In the following figure, we have a SEQ pointer pointing to the second element in an array of four structures of type τ. We also show a SAFE pointer pointing to data of type τ' within one of these structures. We require that the type τ' match the corresponding portion of the type τ according to the physical type equality rules that we will discuss later.



Finally, consider the WILD pointer. The representation of this pointer kind is more complicated than that of a SEQ pointer. WILD pointers come equipped with a base field and a pointer field. The base field points to the start of a dynamically typed area, where there is a length field indicating the size of the area. At the end of the area is a set of tag bits that indicate which words in the dynamically typed area contain CCured's base pointers.



In this example, we show a WILD pointer addressing the second word in a dynamically typed area of four words. The first word contains the integer 7, the second and third words contain a WILD pointer, and the final word contains the integer 5. The tag bits at the end of the area indicate which words contain a valid base pointer; in this example, the third bit has the value one because the third word in the dynamically typed area is a valid base pointer that can be trusted to adhere to our invariants. Note that a dynamically typed area cannot contain SAFE or SEQ pointers.

Expression	Typing premises	Run-time checks and translation
Memory reads		
$*x$	$x : \tau * \texttt{SAFE}$	$\mathsf{assert}(x.p \neq \texttt{null});\ *(x.p)$
$*x$	$x : \tau * \texttt{SEQ}$	$\mathsf{assert}(x.b \leq x.p \leq x.e - \texttt{sizeof}(\tau));\ *(x.p)$
$*x$	$x : \texttt{int} * \texttt{WILD}$	$\mathsf{assert}(x.b \neq \texttt{null});\ \mathsf{assert}(x.b \leq x.p \leq x.b + \mathsf{len}(x.b) - 1);\ *(x.p)$
$*x$	$x : \tau * \texttt{WILD} * \texttt{WILD}$	$\mathsf{assert}(x.b \neq \texttt{null});\ \mathsf{assert}(x.b \leq x.p \leq x.b + \mathsf{len}(x.b) - 2);$ $\mathsf{assert}(\mathsf{tag}(x.b, x.p + 1) == 1);\ *(x.p)$
Memory writes		For all $*x = y$, check $*x$ as above, except omit the tag check. The type of y must match that of $*x$.
$*x = y$	$x : \texttt{int} * \texttt{SAFE}$	$*(x.p) = y$
$*x = y$	$x : \texttt{int} * \texttt{SEQ}$	$*(x.p) = y$
$*x = y$	$x : \tau * q * \texttt{SAFE}$	$\mathsf{assert}(\mathsf{NotStackPointer}(y));\ *(x.p) = y$
$*x = y$	$x : \tau * q * \texttt{SEQ}$	$\mathsf{assert}(\mathsf{NotStackPointer}(y));\ *(x.p) = y$
$*x = y$	$x : \texttt{int} * \texttt{WILD}$	$\mathsf{tag}(x.b, x.p) = 0;\ *(x.p) = y$
$*x = y$	$x : \tau * \texttt{WILD} * \texttt{WILD}$	$\mathsf{assert}(\mathsf{NotStackPointer}(y));\ \mathsf{tag}(x.b, x.p) = 0;$ $\mathsf{tag}(x.b, x.p + 1) = 1;\ *(x.p) = y$

Fig. 3. CCured typing rules for reads and writes. For each kind of expression shown in the left column, the middle column shows the typing premises that make the expression well-typed in CCured, and the right column shows the instrumentation that is added. All arithmetic in the right column is integer arithmetic.

For the remainder of this section we will use the following function as a running example. Given an array of pointers a, the function copies a[10] into a[0], and then into a[1], etc. The value a[10] is then returned as a FILE*. Note that this program is unsafe if the array a has fewer than 11 elements, and uses of the return value of this function may be unsafe if a was not actually an array of FILE*s at run-time.

```
FILE * WILD f(int * WILD * SEQ a) {
  int * WILD * SEQ x;
  int * WILD * SAFE y = a + 10;
  for (x = a ; x < y ; x++)
    *x = *y;
  return (FILE * WILD) *y;
}
```

2.2 Type Checking and Run-Time Checks

The CCured typing rules and the run-time checks for memory operations are shown in Figure 3. For each form of expression shown in the left column of the figure, the middle column shows a number of alternative typing premises under which the expression is well typed. The right column shows what run-time checks CCured adds in each case and how it translates the expression in the left column.

In the case of memory reads, we first check whether the pointer is null. For SEQ and WILD pointers we perform a bounds check.[1] In addition, when reading a pointer through a WILD pointer, we must check the tag bits to verify that the stored pointer has not been altered. The notation len(b) refers to the length of a

[1] For SEQ pointers, if the bounds check passes then x.b $\neq$ null, because x.b = null if and only if x.e = null (Section 4). Therefore the null check for SEQ pointers is not explicitly performed.

Expression	Typing premises	Run-time checks and translation
Allocation		
$(\tau * \texttt{SAFE})\texttt{malloc}(n)$		$\textsf{assert}(n \geq \texttt{sizeof}(\tau));\ \{\ p = \texttt{zeroed\_malloc}(n)\ \}$
$(\tau * \texttt{SEQ})\texttt{malloc}(n)$		$\{\ p = \texttt{zeroed\_malloc}(n),\ b = p,\ e = p + n\ \}$
$(\tau * \texttt{WILD})\texttt{malloc}(n)$		$\texttt{let}\ m = \texttt{zeroed\_malloc}(1 + n + \lceil n/\texttt{bitsperword}\rceil);$ $m[0] = n;\ \{\ p = m + 1,\ b = m + 1\ \}$
Type casts		
$(\texttt{int})x$	$x : \tau * \texttt{SAFE}$	$x.p$
$(\texttt{int})x$	$x : \tau * \texttt{SEQ}$	$x.p$
$(\texttt{int})x$	$x : \tau * \texttt{WILD}$	$x.p$
$(\tau' * \texttt{SAFE})x$	x is the literal 0	$\{\ p = \texttt{null}\ \}$
$(\tau' * \texttt{SEQ})x$	$x : \texttt{int}$	$\{\ p = x,\ b = \texttt{null},\ e = \texttt{null}\ \}$
$(\tau' * \texttt{WILD})x$	$x : \texttt{int}$	$\{\ p = x,\ b = \texttt{null}\ \}$
$(\tau' * \texttt{WILD})x$	$x : \tau * \texttt{WILD}$	x
$(\tau' * \texttt{SAFE})x$	$x : \tau * \texttt{SAFE},\ \tau \approx \tau'$	x
$(\tau' * \texttt{SEQ})x$	$x : \tau * \texttt{SEQ},\ \tau \approx \tau'$	x
$(\tau' * \texttt{SAFE})x$	$x : \tau * \texttt{SEQ},\ \tau \approx \tau'$	$\textsf{assert}(x.p == \texttt{null}\ \Vert$ $x.b \leq x.p \leq x.e - \texttt{sizeof}(\tau'));\ \{\ p = x.p\ \}$
$(\tau' * \texttt{SEQ})x$	$x : \tau * \texttt{SAFE},\ \tau \approx \tau'$	$\{\ p = x.p,\ b = x.p,\ e = x.p + \texttt{sizeof}(\tau)\ \}$
Arithmetic (including aggregate access)		
$x_1 + x_2$	$x_1 : \tau * \texttt{SEQ},\ x_2 : \texttt{int}$	$\{\ p = x_1.p + x_2 * \texttt{sizeof}(\tau),\ b = x_1.b,\ e = x_1.e\ \}$
$x_1 + x_2$	$x_1 : \tau * \texttt{WILD},\ x_2 : \texttt{int}$	$\{\ p = x_1.p + x_2 * \texttt{sizeof}(\tau),\ b = x_1.b\ \}$
$\&(x \rightarrow f_2) : \tau_2 * \texttt{SAFE}$	$x : \texttt{struct}\{\tau_1 f_1;\quad \tau_2 f_2;\} * \texttt{SAFE}$	$\textsf{assert}(x.p \neq \texttt{null});\ \{p = \&(x.p \rightarrow f_2)\}$
$\&(x \rightarrow f_2) : \tau_2 * \texttt{SAFE}$	$x : \texttt{struct}\{\tau_1 f_1;\quad \tau_2 f_2;\} * \texttt{SEQ}$	$\textsf{assert}(x.b \leq x.p \leq x.e - \texttt{sizeof}(\tau_1) - \texttt{sizeof}(\tau_2));$ $\{\ p = \&(x.p \rightarrow f_2)\ \}$
$\&(x \rightarrow f_2) : \tau_2 * \texttt{WILD}$	$x : \texttt{struct}\{\tau_1 f_1;\quad \tau_2 f_2;\} * \texttt{WILD}$	$\{\ p = \&(x.p \rightarrow f_2),\ b = x.b\ \}$

Fig. 4. CCured typing rules for casts, arithmetic, and aggregate accesses. For each kind of expression shown in the left column, the middle column shows the typing premises that make the expression well-typed in CCured, and the right column shows the instrumentation that is added. All arithmetic in the right column is integer arithmetic.

dynamically typed area pointed to by b, and tag(b, p) denotes the tag corresponding to the word pointed to by p inside the dynamically typed area pointed to by b. In our example, line 5 (“*x = *y;”) contains a read from int ∗ WILD ∗ SAFE y. At run-time we ensure that y.p is not null.

For memory writes, we perform the same checks as for reads, and additionally, we check that we do not store a stack pointer. This latter check is a conservative way to prevent dangling references to a stack frame of a function after the function returns. When writing into dynamically typed areas, the tag bits must be updated to reflect the type of what is written; when a pointer is written into such an area, we set the bits corresponding to the stored pointer and base fields to zero and one, respectively. When an integer is written, we clear the tag bit for the written word, thus invalidating any previously stored base field. This scheme maintains the invariant that the tag bit for a word is set to one if and only if the word contains a valid base pointer. In our example, line 5 (“*x = *y;”) contains a write to int ∗ WILD ∗ SEQ x. At run-time we ensure that $\texttt{x.b} \leq \texttt{x.p} \leq \texttt{x.e} - \texttt{sizeof(int} * \texttt{WILD)}$ and that *y is not a stack pointer.

The first six lines in the “Type Casts” section of Figure 4 describe the rules for casts between pointers and integers. The first three lines show that any kind of pointer can be cast to an integer. The fourth line shows that only the integer zero (i.e., the null pointer) can be cast to a SAFE pointer. The fifth and sixth lines show that integers can be cast to SEQ and WILD pointers with the restriction that the base and end fields are set to null. In conjunction with the memory read checks described earlier, this restriction ensures that SEQ and WILD pointers obtained from integers cannot be dereferenced. In our experience, casts from pointers to integers and back to pointers are actually quite rare in C programs.

In order to describe our rules for casts between pointers, we must define a basic notion of physical type equality ($\approx$). The intuition for this approach is that two CCured types should be considered equal if they are laid out in memory in the same way. Thus, in all type comparisons in the rest of this article, we shall consider a CCured type τ to be a list of atomic types α obtained by unrolling and flattening arrays and structures, as specified by the ϕ function defined below. The ϕ function uses the nil and :: constructors for the empty and nonempty lists, respectively; in addition, we use an append operator, written @.

$$\begin{array}{lcl}
\phi(\texttt{void}) & = & \mathit{nil} \\
\phi(\texttt{int}) & = & \texttt{int} :: \mathit{nil} \\
\phi(\tau * q) & = & \tau * q :: \mathit{nil} \\
\phi(\texttt{struct\{}\ \tau_1 f_1; \ldots \tau_n f_n;\ \texttt{\}}) & = & \phi(\tau_1) \mathbin{@} \ldots \mathbin{@} \phi(\tau_n) \\
\phi(\tau[n]) & = & \underbrace{\phi(\tau) \mathbin{@} \ldots \mathbin{@} \phi(\tau)}_{n \text{ times}}
\end{array}$$

For now, we ignore structure padding and alignment, assume that structs are associative, and pretend all fields are represented contiguously in memory; however, Section 5.7 fills this gap. Our actual implementation unrolls arrays lazily; it only expands types as much as needed for the type comparison operations that we will describe next.

Physical type equality, written $\approx$, is defined as element-wise equality over lists of atomic types, ignoring the static types of WILD pointers. Formally, the definition of physical type equality is as follows:

$$\frac{}{\mathit{nil} \approx \mathit{nil}} \qquad \frac{\alpha \approx \alpha' \quad \tau \approx \tau'}{\alpha :: \tau \approx \alpha' :: \tau'}$$

$$\frac{}{\texttt{int} \approx \texttt{int}} \qquad \frac{\tau_1 \approx \tau_2}{\tau_1 * q \approx \tau_2 * q} \qquad \frac{}{\tau_1 * \texttt{WILD} \approx \tau_2 * \texttt{WILD}}$$

In Figure 4, casts among SAFE and SEQ pointers are governed by physical equality of their base types. For example, a SAFE pointer to an array of two ints can be cast to a SAFE pointer to a structure containing two ints, and a SEQ pointer to int (a pointer to an array of ints) can be cast to a SAFE pointer to int (a pointer to a single element). These restrictions will be revised in Section 3.1 to account for physical subtyping. In our running example, line 3 contains an implicit cast from the SEQ pointer a+10 to the SAFE pointer y: "int * WILD * SAFE y = a + 10." Since the underlying types are equal, we ensure at run time that either a.p + 10 == null or that a.b $\leq$ a.p + 10 $\leq$ a.e − sizeof(int * WILD). This run-time check allows us to maintain our invariant that the safe pointer y is either null or a valid pointer to an element of its base type.

Finally, arbitrary pointer arithmetic is allowed only for SEQ and WILD pointers. In our example, lines 3 and 4 contain pointer arithmetic ("a + 10" and "x++")

and both pointers, a and x, are SEQ. However, as a more controlled form of arithmetic, CCured supports the creation of pointers to fields of structures, which is a common operation in many C programs. Consider the representative case $\&(x \rightarrow f_2)$, where x is a pointer to a structure. If x is a SAFE pointer, then we require that x be nonnull; if x were null this expression would yield a SAFE pointer that is neither valid nor null. If x is a SEQ pointer, we must first convert the pointer to a SAFE one (hence the bounds check), and then we can obtain a SAFE pointer to the second field.

2.2.1 *Initializing Pointers.* There are two ways to create new pointer values in C: programs can call malloc, or they can take the address of a local or global variable. In CCured, these operations must be modified slightly so that they initialize metadata for the newly created pointer.

Figure 4 shows how pointers are created using malloc. For SEQ pointers, this is as simple as setting the bounds information for the pointer. For WILD pointers, we must allocate extra memory to hold the length field and tag bits, as shown in the diagram in Section 2.1.

Our simplified language does not include an address-of operator "&," but extending the initialization shown in Figure 4 is straightforward. For SEQ pointers, we take the address of the variable and then set the bounds metadata appropriately. However, if we create a WILD pointer to a variable, we must also change the layout of that variable when it is allocated, to make room for the length and tag bits.

2.3 Pointer Kind Inference

The CCured type system assumes that pointers are already annotated with pointer qualifiers. In order to use CCured on existing C programs without such annotations, we use a whole-program pointer-kind inference. We associate a qualifier variable with each syntactic occurrence of the pointer-type constructor (written $*$) and with the address of every variable. We then scan the program, collecting a set of constraints on these qualifier variables. Finally, we solve the system of constraints to obtain an assignment of qualifiers to qualifier variables such that the resulting program type checks in the CCured system.

The overall goal of inference is to find as many SAFE pointers as possible and then to find as many SEQ pointers as possible. Simply making all qualifiers WILD would yield a well-typed solution, but SAFE and SEQ pointers are preferred for performance and interoperability.

(1) *Constraint Collection.* We collect constraints using a modified typing judgment written $\Gamma \vdash e : \tau \mapsto C$. This judgment says that by scanning the expression e in context Γ (a mapping from variables to atomic types), we infer type τ along with a set of constraints C. We also use the auxiliary judgments τ_1 castto $\tau_2 \mapsto C$ to collect constraints corresponding to the conversion when casting an object of type τ_1 to an object of type τ_2.[2] Any solution to the set of constraints C assigns a pointer kind to every qualifier

[2] For simplicity, we assume that all such casts are made explicit; in fact, the CIL [Necula et al. 2002b] infrastructure that we use to implement CCured will add such casts automatically.

Expressions:

$$\frac{\Gamma \vdash e_1 : \tau * q \mapsto C_1 \quad \Gamma \vdash e_2 : \texttt{int} \mapsto C_2}{\Gamma \vdash e_1 + e_2 : \tau * q \mapsto C_1 \cup C_2 \cup \{q \neq \texttt{SAFE}\}}\ \textsf{arith} \qquad \frac{\Gamma \vdash e : \tau_1 \mapsto C_1 \quad \tau_1\ \texttt{castto}\ \tau_2 \mapsto C_2}{\Gamma \vdash (\tau_2)e : \tau_2 \mapsto C_1 \cup C_2}\ \textsf{cast}$$

$$\frac{}{\Gamma \vdash (\tau * q)0 : \tau * q \mapsto \emptyset}\ \textsf{null} \qquad \frac{\Gamma \vdash e : \tau * q \mapsto C}{\Gamma \vdash (\tau * q)\texttt{malloc}(e) : \tau * q \mapsto C}\ \textsf{malloc} \qquad \frac{\Gamma \vdash e : \tau * q \mapsto C}{\Gamma \vdash * e : \tau \mapsto C}\ \textsf{ref}$$

$$\frac{\Gamma \vdash e_1 : \tau * q \mapsto C_1 \quad \Gamma \vdash e_2 : \tau \mapsto C_2}{\Gamma \vdash *e_1 = e_2 : \tau \mapsto C_1 \cup C_2}\ \textsf{assign}$$

Convertibility:

$$\frac{}{\tau\ \texttt{castto int} \mapsto \emptyset}\ \textsf{toInt} \qquad \frac{}{\texttt{int castto}\ \tau * q \mapsto \{q \neq \texttt{SAFE}\}}\ \textsf{fromInt}$$

$$\frac{}{\begin{array}{ll}\tau_1 * q_1\ \texttt{castto}\ \tau_2 * q_2 \mapsto & \{\ q_1 = q_2 = \texttt{WILD}\ \vee\ \tau_1 \approx \tau_2\ \} \ \cup \\ & \{\ q_1 = \texttt{WILD} \iff q_2 = \texttt{WILD}\ \} \ \cup \\ & \{\ q_2 = \texttt{SEQ} \implies q_1 = \texttt{SEQ}\ \}\end{array}}\ \textsf{castPtr}$$

Fig. 5. Constraint generation rules for CCured pointer kind inference. Recall that zero (null) can be cast to any pointer kind, while other integers may be cast only to SEQ or WILD pointers so that we can prevent them from being dereferenced.

variable in the expression e such that the resulting program type-checks according to Figure 3 and Figure 4. The rules for the constraint collection judgments are shown in Figure 5.

In addition, for each type of the form $\tau * q' * q$ (i.e., a pointer to a pointer) or struct{ ... $\tau * q' f$; ... } $* q$ (i.e., a pointer to a structure containing a pointer), we collect a POINTSTO constraint $q = \texttt{WILD} \implies q' = \texttt{WILD}$. This constraint captures the requirement that all WILD pointers point only to dynamically typed areas.

After constraint generation, we end up with a set containing the following five kinds of constraints:

ARITH:	$q \neq \texttt{SAFE}$
BOUNDS:	$q = \texttt{SEQ} \implies q' = \texttt{SEQ}$
POINTSTO:	$q = \texttt{WILD} \implies q' = \texttt{WILD}$
CAST:	$q = \texttt{WILD} \iff q' = \texttt{WILD}$
TYPEEQ:	$q = q' = \texttt{WILD}\ \vee\ \tau_1 \approx \tau_2$

Consider again our running example, this time with the pointer qualifier values erased. In their place, we have pointer qualifier variables, each designated by a subscript q_i.

```
1 : FILE *q1 f(int *q2 *q3 a) {
2 :   int *q4 *q5 x;
3 :   int *q6 *q7 y = a + 10;
4 :   for (x = a ; x < y ; x++)
5 :     *x = *y;
6 :   return (FILE *q8) *y;
7 : }
```

The arithmetic on line 3 will generate the ARITH constraint $q_3 \neq \texttt{SAFE}$ using the arith rule. The increment on line 4 will generate $q_5 \neq \texttt{SAFE}$. The

$$\frac{}{\vdash nil \approx nil \mapsto \emptyset} \qquad \frac{\vdash \alpha_1 \approx \alpha_2 \mapsto C_1 \quad \vdash \tau_1 \approx \tau_2 \mapsto C_2}{\vdash \alpha_1 :: \tau_1 \approx \alpha_2 :: \tau_2 \mapsto C_1 \cup C_2}$$

$$\frac{}{\vdash \texttt{int} \approx \texttt{int} \mapsto \emptyset} \qquad \frac{\vdash \tau_1 \approx \tau_2 \mapsto C}{\vdash \tau_1 * q_1 \approx \tau_2 * q_2 \mapsto C \cup \{\ q_1 = q_2\ \}}$$

$$\frac{}{\vdash \tau_1 * q_1 \approx \tau_2 * q_2 \mapsto \{\ q_1 = q_2 = \texttt{WILD}\ \}}$$

Fig. 6. Constraint generation judgment based on physical equality of lists of atomic types.

cast implicit in the assignment on line 3 requires $\texttt{int} * q_2 * q_3$ `castto` $\texttt{int} * q_6 * q_7$, which generates $\{q_3 = q_7 = \texttt{WILD} \ \vee\ \texttt{int} * q_2 \approx \texttt{int} * q_6\}$, $\{q_3 = \texttt{WILD} \iff q_7 = \texttt{WILD}\}$, and $\{q_7 = \texttt{SEQ} \implies q_3 = \texttt{SEQ}\}$. The other assignments are similar. The remaining cast on line 6 generates the constraints $\{q_6 = q_8 = \texttt{WILD} \ \vee\ \texttt{int} \approx \texttt{FILE}\}$, $\{q_6 = \texttt{WILD} \iff q_8 = \texttt{WILD}\}$, and $\{q_8 = \texttt{SEQ} \implies q_6 = \texttt{SEQ}\}$. POINTSTO constraints like $q_5 = \texttt{WILD} \implies q_4 = \texttt{WILD}$ will also be generated based on the type structure of the program.

(2) *Constraint Normalization.* The next step is to normalize the generated constraints into a simpler form. First, CAST constraints are converted into pairs of POINTSTO constraints. Next, we consider POINTSTO constraints. Since these constraints are conditional constraints, we can ignore them as long as the qualifier q on the left is unknown. If this qualifier becomes WILD, we add the constraint "$q' = \texttt{WILD}$" to the system of constraints. If q remains unknown at the end of the normalization process, we will make it SAFE or SEQ. BOUNDS constraints will be handled in a manner similar to POINTSTO constraints.

The TYPEEQ constraints cannot be normalized as easily. At the first sight these are conditional constraints for which the solver might have to perform an expensive case analysis. Fortunately, this is not necessary. Instead, the constraint $q_1 = q_2 = \texttt{WILD} \ \vee\ \tau_1 \approx \tau_2$ is simplified as follows:

$$\begin{cases} C & \text{if } \vdash \phi(\tau_1) \approx \phi(\tau_2) \mapsto C \\ q_1 = q_2 = \texttt{WILD} & \text{otherwise} \end{cases}$$

Essentially, the two types are converted into lists of atomic types and are compared element-wise. The rules for collecting physical type equality constraints for two types are given in Figure 6. The first two rules deal with list operations. The third rule handles the case for the atomic type `int`. The last two rules handle casts between pointers. We use the final rule, which makes the pointers WILD, only when the premise of the other rule cannot be applied.

If the comparison succeeds, it means that two types contain pointers in the same corresponding positions and the resulting constraints C (consisting only of simple equality constraints $q_1' = q_2'$) ensures that such corresponding pointers will be given identical pointer kinds. If we fail to derive the judgment $\vdash \tau_1 \approx \tau_2 \mapsto C$, then under no instantiation of pointer kind variables with pointer kinds will we have $\tau_1 \approx \tau_2$; thus, we replace the whole TYPEEQ constraint with $q_1 = q_2 = \texttt{WILD}$.

The key insight that leads to a linear time algorithm is that this procedure for simplifying the TYPEEQ conditional constraints results in a set of constraints that is not only sufficient but also necessary. Clearly the resulting constraints are sufficient, because if any of the two cases are satisfied then the disjunctive TYPEEQ constraint is satisfied. To argue the converse we must consider the cases when each of the disjuncts of the TYPEEQ constraint is satisfied in turn. If the first disjunct $q_1 = q_2 = \mathtt{WILD}$ is satisfied and we can derive $\phi(\tau_1) \approx \phi(\tau_2) \mapsto C$ then it must be that all pointer kinds in τ_1 and τ_2 are WILD (due to the POINTSTO constraints); hence they are equal and C is satisfied. If the second disjunct is satisfied then we are guaranteed to be able to derive $\phi(\tau_1) \approx \phi(\tau_2) \mapsto C$ and clearly C is satisfied.

In our running example, we have the TYPEEQ constraint $\{q_6 = q_8 = \mathtt{WILD} \vee \mathtt{int} \approx \mathtt{FILE}\}$. We will be unable to derive a judgment $\vdash \mathtt{int} \approx \mathtt{FILE}$, so we will simplify it to $\{q_6 = q_8 = \mathtt{WILD}\}$. Intuitively, the cast between int and FILE is a bad cast that will introduce WILD pointers into the system. However, we will be able to derive a judgment $\vdash \mathtt{int} * q_2 \approx \mathtt{int} * q_6$ from the constraint $\{q_3 = q_7 = \mathtt{WILD} \vee \mathtt{int} * q_2 \approx \mathtt{int} * q_6\}$. This judgment will generate the constraint $\{q_2 = q_6\}$, which will replace the original TYPEEQ constraint.

After simplifying all TYPEEQ constraints, the normalized system has only the following kinds of constraints:

ARITH:	$q \neq \mathtt{SAFE}$
BOUNDS:	$q = \mathtt{SEQ} \implies q' = \mathtt{SEQ}$
POINTSTO:	$q = \mathtt{WILD} \implies q' = \mathtt{WILD}$
ISDYN:	$q = \mathtt{WILD}$
EQ:	$q = q'$

(3) *Constraint Solving.* The final step in our algorithm is to solve the remaining set of constraints. The algorithm is quite simple:

3.1 Propagate the ISDYN constraints using the constraints EQ and POINTSTO. This step propagates the WILD qualifier as far as necessary; all qualifiers not assigned by this step can be SEQ or SAFE.

3.2 All qualifier variables involved in ARITH constraints are set to SEQ, and this information is propagated using EQ and BOUNDS constraints. Note that BOUNDS constraints propagate SEQ qualifiers against the flow of assignments and casts in order to ensure that a pointer that needs bounds will carry them from its source.

3.3 We make all the other variables SAFE.

Essentially, we start by finding the minimum number of WILD qualifiers. Among the remaining qualifiers, we find those on which pointer arithmetic is performed, and we make them SEQ. The remaining qualifiers are SAFE. This solution is the best one possible in terms of maximizing the number of SAFE and SEQ pointers.

In our example, we start with ISDYN constraints $\{q_8 = q_6 = \mathtt{WILD}\}$. Propagation via EQ constraints yields $\{q_1 = q_2 = q_4 = \mathtt{WILD}\}$. We then consider the ARITH constraints $\{q_7 \neq \mathtt{SAFE}\}$ and $\{q_5 \neq \mathtt{SAFE}\}$. Since have found all WILD pointers, this means $\{q_7 = q_5 = \mathtt{SEQ}\}$. We propagate SEQ back using BOUNDS constraints to obtain $\{q_3 = \mathtt{SEQ}\}$. All other pointer qualifiers are SAFE.

The whole type inference process is linear in the size of the program. A linear number of qualifier variables is introduced (one for each syntactic

```
struct Figure {
  double (*area)(struct Figure * obj);
};
struct Circle {
  double (*area)(struct Figure * obj);
  double radius;
} *c;
double Circle_area(struct Figure *obj) {
  struct Circle *cir = (struct Circle*)obj;    // downcast
  return PI * cir->radius * cir->radius;
}
c->area((struct Figure *)c);                  // upcast
```

Fig. 7. Example code fragment illustrating subtyping, upcasts, and downcasts. Circle is meant to be a subtype of Figure.

occurrence of a pointer type constructor), and then a linear number of constraints is created (one for each cast or memory read or write in the program). During the simplification of the TYPEEQ constraints, the number of constraints can get multiplied by the maximum nesting depth of a qualifier in a type. Finally, constraint solving is linear in the number of constraints.

3. COPING WITH CASTS

Many C programs make heavy use of casts between pointer types. We must try to verify statically that most of the type casts are compatible with the CCured type system; otherwise, we will have to classify many pointers as WILD, which creates performance and compatibility problems. Notice that WILD pointers create more challenging compatibility problems than the other pointer kinds because the memory area to which they point requires a special layout. This problem is exacerbated by the extensive spreading of the WILD qualifiers. For example, if a FILE * value is involved in a bad cast, it becomes WILD and also requires the return value of the fopen function to become WILD. To support this kind of fopen we would need to change the layout of the statically allocated FILE structures in the standard C library to include the necessary tags.

As an easy escape hatch, CCured allows the programmer to assert that an otherwise bad cast can be trusted. This mechanism is a controlled loss of soundness and assumes an external review of those casts. Still, this approach has practical value in focusing a code review when the number of such casts is relatively small. One standard application of such a trusted cast is a custom allocator in which a portion of an array of characters is cast to an object.

Fortunately, there are many situations in which we can reason effectively about casts between unequal types. For example, consider the code fragment in Figure 7, which contains object-oriented style subtype polymorphism. In this example, Circle is meant to be a subtype of Figure. Both structures include a function pointer, which is set to Circle_area in the case of circles. The program can compute the area of any figure by invoking the function pointer as shown at the end of the code fragment (a form of dynamic dispatch). According to the strict classification of types from before, there are two bad casts: one in the body of Circle_area where the input argument is cast to Circle * (a *downcast* in

Expression	Typing premises	Run-time checks and translation
$(\tau' * \texttt{SAFE})x$	$x : \tau * \texttt{SAFE},\ \tau \lesssim \tau'$	x
$(\tau' * \texttt{SEQ})x$	$x : \tau * \texttt{SEQ},\ \tau[n] \approx \tau'[n']$	x
$(\tau' * \texttt{SAFE})x$	$x : \tau * \texttt{SEQ},\ \tau[n] \lesssim \tau'$	$\mathsf{assert}(x.p == \texttt{null}\ \Vert$ $x.b \leq x.p \leq x.e - \texttt{sizeof}(\tau'));\ \{\ p = x.p\ \}$
$(\tau' * \texttt{SEQ})x$	$x : \tau * \texttt{SAFE},\ \tau'[n'] \lesssim \tau$	$\{\ p = x.p,\ b = x.p,\ e = x.p + \texttt{sizeof}(\tau)\ \}$

Fig. 8. CCured physical subtyping rules for casts. For an expression in the left column, the middle column shows the typing premises, and the right column shows the instrumentation that is added. The variables $n, n' > 0$ are the smallest integers such that $n \cdot \texttt{sizeof}(\tau) = n' \cdot \texttt{sizeof}(\tau')$.

the subtype hierarchy), and one in the invocation of the `area` method at the end of the code fragment (an *upcast*).

Siff et al. [1999] observed that a large fraction of the casts between unequal types in real programs are either upcasts or downcasts, and our experiments support this claim. In particular, we have observed that around 63% of casts are between identical types. Of the remaining casts, about 93% are safe upcasts and 6% are downcasts. Less than 1% of all casts fall outside of these categories; these casts must still be considered bad even in the presence of mechanisms that handle downcast and upcasts. These numbers were taken from the `bind` benchmark; other large programs in Section 8 were similar. In the rest of this section, we describe two mechanisms, one for dealing with upcasts and one for downcasts, with the overall goal of reducing drastically the number of casts that CCured considers bad.

3.1 Upcasts and Physical Subtyping

An upcast is a cast from type $\tau\ *$ to type $\tau'\ *$ when the aggregate τ' is laid out in memory exactly as a prefix of the layout of the aggregate τ. This relationship between types τ and τ' is called *physical subtyping* and has been shown previously to be important for understanding the typing structure of C programs [Chandra and Reps 1999; Siff et al. 1999].

We define the physical subtyping relation $\tau \lesssim \tau'$ by requiring that the type τ be physically equal to the concatenation of the type τ' and some other (possibly empty) list of atomic types τ'':

$$\tau \lesssim \tau' \ \overset{\text{def}}{\iff}\ \exists \tau''.\ \tau \approx \tau' \mathbin{@} \tau''.$$

With these definitions, the more relaxed CCured typing rules for casts are shown in Figure 8. These rules replace the original rules in Figure 4.

Our notion of physical subtyping for `SAFE` pointers is different from that of previous work; specifically, CCured differs in its handling of `void*` and of pointer arithmetic. In previous work [Chandra and Reps 1999; Siff et al. 1999], `void*` was allowed in the smaller aggregate in any position where a regular pointer was present in the larger one. This approach is unsound; instead, `void` should be considered to be the empty structure (or empty list of atomic types), and any type should be considered a physical subtype of `void`. As a result, we can safely cast a pointer to any type into `void*`. However, when we try to use the resulting `void*`, we have to cast it to some other pointer type first; this downcast operation is handled later in this section.

Physical subtyping must also be modified in the presence of pointer arithmetic; we cannot use simple width subtyping as with `SAFE` pointers. For example, it is not safe to cast a pointer `cs` of type `struct Circle * SEQ` to type `struct Figure * SEQ`, because then the memory word residing at address `cs`→`radius` can be accessed as a `double` and also as a function pointer using `((struct Figure * SEQ)cs + 1)`→`area`.

To fix this soundness problem, we require for each type cast on `SEQ` pointers that $\tau'[n'] \approx \tau[n]$ where $n, n' > 0$ are the smallest integers such that $n \cdot \texttt{sizeof}(\tau) = n' \cdot \texttt{sizeof}(\tau')$. Notice that when changing the type of a `SEQ` pointer into another `SEQ` pointer, the representation does not change. Casting between `SEQ` pointers also allows for a robust treatment of multidimensional arrays (in which case either n or n' is typically 1) and is necessary for handling many nontrivial C programs.

3.1.1 *Changes to the Inference Algorithm.* In order to take advantage of physical subtyping, the inference algorithm must be extended to pay special attention to casts. The convertibility constraint generation rule for pointer casts is replaced with a rule that uses physical subtyping instead of physical equality:

$$\frac{}{\begin{array}{rl} \tau * q \ \texttt{castto}\ \tau' * q' \mapsto & \{\, q = q' = \texttt{WILD} \ \vee\ \tau \lesssim \tau' \,\} \ \cup \\ & \{\, q = \texttt{WILD} \iff q' = \texttt{WILD} \,\} \ \cup \\ & \{\, q' = \texttt{SEQ} \implies q = \texttt{SEQ} \,\} \end{array}}$$

We call a constraint of the form $q = q' = \texttt{WILD} \ \vee\ \tau \lesssim \tau'$ a `TYPESUB` constraint. We normalize such a constraint by finding a derivation of the form $\vdash \tau \lesssim \tau' \mapsto C$, using the following rules:

$$\frac{}{\vdash \tau \lesssim nil \mapsto \emptyset} \qquad \frac{\vdash \alpha \approx \alpha' \mapsto C_1 \quad \vdash \tau \lesssim \tau' \mapsto C_2}{\vdash \alpha :: \tau \lesssim \alpha' :: \tau' \mapsto C_1 \cup C_2}$$

The first rule allows any type to be a physical subtype of `void`. The second rule allows physical subtypes to share a common prefix.

Notice that a `TYPESUB` constraint allows only the following combinations of casts: `WILD`-to-`WILD`, `SAFE`-to-`SAFE`, `SEQ`-to-`SEQ`, and `SEQ`-to-`SAFE`. In particular, it disallows the cast `SAFE`-to-`SEQ`. This approach might be overly conservative (since the CCured type system allows such casts), but our experience shows that, in the absence of user annotations, a `SEQ` constraint should be propagated against the flow of data to ensure that the right bounds are set at the memory allocation location. In order to simplify a `TYPESUB` constraint, we first look for a derivation of the form $\vdash \tau[n] \approx \tau'[n'] \mapsto C_1$, where n and n' can be computed using the size of the types τ and τ'.

—If such a derivation exists, we then check to see if $\texttt{sizeof}(\tau) \geq \texttt{sizeof}(\tau')$, which in this context implies $\tau \lesssim \tau'$. If this relation holds, then we replace `TYPESUB` with C_1. This case is the least constraining one; since the typing premises for all four possibilities in Figure 8 are satisfied, we can allow any combination of `SAFE` or `SEQ` qualifiers for q and q'.

If instead $\texttt{sizeof}(\tau) < \texttt{sizeof}(\tau')$, then we replace `TYPESUB` with $C_1 \cup \{\, q \neq \texttt{SAFE} \,\}$. In this case, the typing premises for a `SAFE`-to-`SAFE` cast do not hold, and we have seen that the `SAFE`-to-`SEQ` possibility is ruled out as well.

—If no such derivation exists, we look instead for a derivation of the form $\vdash \tau \lesssim \tau' \mapsto C_2$. If we find one, we replace TYPESUB with $C_2 \cup \{\, q' \neq \mathtt{SEQ} \,\}$, a new NOTSEQ constraint whose handling we explain below. In this case, the typing premises for a SAFE-to-SAFE cast are satisfied, as are those for a SEQ-to-SAFE cast ($\tau \lesssim \tau' \implies \tau[n] \lesssim \tau'$). However, a SEQ-to-SEQ cast is not allowed (not finding the first derivation ruled out its premises), nor is a SAFE-to-SEQ one, as explained above.

If we cannot find a derivation, we conservatively replace the TYPESUB with $\{\, q = q' = \mathtt{WILD} \,\}$. A SAFE-to-SEQ cast might still be possible if $\tau'[n'] \lesssim \tau$, but because we propagate SEQ backwards using the constraint $\{\, q' = \mathtt{SEQ} \implies q = \mathtt{SEQ} \,\}$ we will never infer one.

We modify the constraint solving algorithm to handle NOTSEQ constraints of the form $q \neq \mathtt{SEQ}$ by adding a step between 3.2 and 3.3 (see Section 2.3) that checks to see if any such q has been assigned a value SEQ. If such an assignment has occurred, then we set $q = \mathtt{WILD}$, remove the NOTSEQ constraint, and return to step 3.1. In our experience, this step is very rarely used. With these changes, the CCured inference algorithm can now automatically recognize upcasts.

3.2 Downcasts and Run-Time Type Information

A downcast is a cast from a type $\tau\, *$ to $\tau'\, *$ when τ' is a physical subtype of τ. One example of a downcast is the cast in the body of the Circle_area function shown in the previous section. Such examples seem to arise often in large programs when C programmers try to use subtype polymorphism and dynamic dispatch to achieve an object-oriented structure for their programs.

Another frequent occurrence of a downcast is a cast from void* to any other pointer type. An interesting result of our experiments is that only a small percentage of uses of void* can be attributed to implementations of parametric polymorphism (e.g., arrays whose elements all have the same dynamic type). More often it seems void* is used for implementing the more expressive subtype polymorphism (e.g., arrays whose elements have distinct types that are all subtypes of void*).

If we classify all downcasts as bad casts, we essentially ignore static type information, which is undesirable. Instead, we extend the CCured type system with a new pointer kind, RTTI, that allows checked downcasts using run-time type information in a manner similar to the checked downcasts in typed object-oriented languages. In the context of CCured, however, we have to answer several questions. First, how should the run-time type information be encoded, and should it be carried with the pointer or stored with the referenced object? Second, what changes are necessary to the CCured inference mechanism to use RTTI pointers in existing C programs?

We represent the run-time type information as nodes in a global tree data structure that encodes the physical subtyping hierarchy of a program. There is a compile-time function, rttiOf, that maps a type to its node in the hierarchy data structure, and a run-time function, isSubtype, that checks whether one node is a physical subtype of another. In addition, we have decided to store the run-time type information with the pointer and not with the referenced

Expression	Typing premises	Run-time checks and translation
$(\tau' * \texttt{RTTI})x$	$x : \tau * \texttt{SAFE},\ \tau \lesssim \tau'$	$\{\ p = x,\ t = \texttt{rttiOf}(\tau)\ \}$
$(\tau' * \texttt{RTTI})x$	$x : \tau * \texttt{RTTI},\ \tau \lesssim \tau'$	x
$(\tau' * \texttt{RTTI})x$	$x : \tau * \texttt{RTTI},\ \tau' \lesssim \tau$	$\texttt{assert}(\texttt{isSubtype}(x.t, \texttt{rttiOf}(\tau')));\ x$
$(\tau' * \texttt{SAFE})x$	$x : \tau * \texttt{RTTI}$	$\texttt{assert}(\texttt{isSubtype}(x.t, \texttt{rttiOf}(\tau')));\ x.p$

Fig. 9. CCured typing rules for casts involving RTTI pointers. For an expression in the left column, the middle column shows the typing premises, and the right column shows the instrumentation that is added.

object, as it is done in object-oriented languages. The main reason in favor of this choice is that C allows pointers to point into the interior of an allocation unit (e.g., to a field of a structure or to an element of an array). In such cases it would have been hard or impossible to insert the run-time type information at a statically known offset in the referenced object. Furthermore, we have observed experimentally with other pointer kinds in CCured that if we change the layout of pointers to objects rather than that of the objects themselves, we increase the likelihood that the transformed code will be compatible with external libraries.

The representation of a pointer of type $\tau * \texttt{RTTI}$ consists of two words, one encoding the pointer value and the other encoding the node in the subtype hierarchy that corresponds to its actual run-time type:

$$\mathsf{Rep}(\tau * \texttt{RTTI}) = \texttt{struct}\{\ \mathsf{Rep}(\tau) * p, \texttt{RttiNode} * t\}.$$

CCured maintains the invariant that such a pointer is either null or otherwise points to a valid object of some type that is a physical subtype of τ. This invariant means that such a pointer can be safely dereferenced just like a τ * pointer if needed; alternatively, it can be cast to some physical subtype of τ with a run-time check.

In Figure 9 we show the necessary changes to the CCured type system and instrumentation. Notice that a cast from SAFE to RTTI must be an upcast and that the original type is recorded in the newly created pointer. Among RTTI pointers we allow both upcasts or downcasts, but in the latter case we check at run-time that the representation invariant is preserved. A similar check is performed when we cast from RTTI to SAFE. The rules for dereferencing RTTI pointers are the same as for SAFE pointers.

3.2.1 *Changes to the Inference Algorithm.* The inference algorithm considers each cast from type $\tau * q$ to type $\tau' * q'$ and collects constraints on the pointer kind variables q and q' as follows:

—If this cast is a downcast ($\tau' \lesssim \tau$) then $q = \texttt{RTTI}$.

—If the base types are physically equal ($\tau \approx \tau'$) then the RTTI kind propagates against the data flow:

$q' = \texttt{RTTI} \implies q = \texttt{RTTI}$.

—If this cast is an upcast ($\tau \lesssim \tau'$), then the RTTI pointer kind propagates against the data flow when the source type has subtypes:

$q' = \texttt{RTTI} \wedge (\exists \tau''.\ \tau'' \lesssim \tau) \implies q = \texttt{RTTI}$.

—Otherwise, this cast is a bad cast and $q = q' = \texttt{WILD}$.

The first two rules identify the downcasts and propagate the requirement for run-time type information to the origin of the pointer. The third inference rule attempts to restrict the backwards propagation of the `RTTI` kind to those types that have subtypes (the existential quantifier ranges over the types actually occurring in the program). If a pointer type does not have subtypes in the program, then the representation invariant of `RTTI` pointers ensures that its static type is the same as its run-time type, and thus the `RTTI` pointer kind is not necessary; instead, we use the pointer kind `SAFE`, which saves both space and time.

For example, consider the following sequence of casts, which uses the types introduced before:

$$\texttt{Circle}{*}\, q_1 \longrightarrow \texttt{Figure}{*}\, q_2 \longrightarrow \texttt{void}{*}\, q_3 \longrightarrow \texttt{Circle}{*}\, q_4.$$

The new inference rules generate constraints that require q_3 to be `RTTI` (due to the downcast from `void*` to `Circle *`) and then will propagate the `RTTI` kind to q_2. However, the `RTTI` kind does not propagate to q_1 since `Circle *` does not have subtypes in the program. The variable q_4 is unconstrained and thus remains `SAFE`.

The `RTTI` pointer kind naturally supports the parametric polymorphism discipline as well as other programming practices common in C programs, such as various flavors of dynamic dispatch and generic data structures. The inference rules associated with this pointer kind are simple, and the results of inference are predictable.

4. TYPE AND MEMORY SAFETY

Figure 3 and Figure 4 give a compact description of a representative fragment of the CCured type system and operational semantics. The type system separates statically typed and dynamically typed pointers, and the operational semantics describe the run-time checks we perform for each pointer operation. In this section we outline the key ideas for a proof of the safety guarantees we obtain for CCured programs that use physical subtyping as described in the previous sections. In order to keep the formalism simple, we do not cover the `RTTI` pointer kind in this section, and we assume that the size of a machine word is 1 byte.

We assume a standard typing judgment $\Gamma \vdash e : \tau$. The environment Γ maps variable names to atomic types. If a derivation $\Gamma \vdash e : \tau$ can be found, then the expression e has type τ. We shall only consider well-typed programs. The typing judgment includes all of the typing premises mentioned in Figures 3, 4, and 8.

As presented, CCured's operational semantics are given in terms of C statements and assertions. In C, values are machine words that can represent either memory locations or integers. In order to better expose the precise costs of using each kind of pointer, we use a low-level representation of addresses as natural numbers and pointers as tuples.

For the purposes of formalizing the operational semantics and proving memory safety, it is useful to consider a mapping Σ from variable names to values, a partial mapping M (the memory) from addresses to values, and a partial mapping W (the type of each memory word) from addresses to atomic types. We require that $Dom(M) = Dom(W)$.

Both the mapping Σ and the mapping Γ are assumed to be provided externally. In our language, only the memory changes during the execution. In particular, W remains constant, and we will not consider memory allocation and deallocation. We consider the following kinds of values:

$$\mathit{Values}\ v \ ::= \ n \ \mid \ \mathit{Safe}(p) \ \mid \ \mathit{Seq}(p, b, e) \ \mid \ \mathit{Wild}(p, b).$$

Every value is either an integer, a `SAFE` pointer, a `SEQ` pointer with bounds information, or a `WILD` pointer into a region with a base and a length (given by $\mathsf{len}(b)$).

The operational semantics are defined by means of two judgments. We write $\Sigma, M \vdash e \Downarrow v, M'$ to say that, in the environment Σ and in the memory state denoted by M, the expression e evaluates to value v and yields a new memory state M'. CCured's analyses and transformations are flow-insensitive, so we assume all expressions terminate. The rules given in Figure 3 and Figure 4, when combined with a formal treatment of C (e.g., Necula et al. [2002b]), give rise to straightforward operational semantics derivation rules.

Recall that we assume we have used ϕ to convert standard C types into lists of atomic types (see Section 1). In order to simplify the presentation we will write $Nth(\tau, i)$ to stand for the atomic type at offset i from the beginning of τ. For example, if $\tau = \texttt{int} :: \texttt{int} :: \tau' * \texttt{SAFE} :: nil$ then $Nth(\tau, 0) = \texttt{int}$ and $Nth(\tau, 2) = \tau' * \texttt{SAFE}$.

Next, we define for each atomic type α the set $\|\alpha\|_W$ of valid values of that type. As the notation suggests, this set depends on the current typing of memory:

$$\begin{aligned}
\|\texttt{int}\|_W &= \mathbb{N},\\
\|\tau * \texttt{SAFE}\|_W &= \{\mathit{Safe}(p) \mid \forall i.\ 0 \le i < \texttt{sizeof}(\tau) \implies W(p+i) \approx Nth(\tau, i)\}\\
&\quad \cup \ \{\mathit{Safe}(0)\},\\
\|\tau * \texttt{SEQ}\|_W &= \{\mathit{Seq}(p, b, e) \mid \forall q.\ b \le q < e \implies\\
&\qquad\qquad W(q) \approx Nth(\tau, (q-p) \bmod \texttt{sizeof}(\tau))\}\\
&\quad \cup \ \{\mathit{Seq}(p, 0, 0) \mid p \in \mathbb{N}\},\\
\|\tau * \texttt{WILD}\|_W &= \{\mathit{Wild}(p, b) \mid \forall q.\ b \le q < b + \mathsf{len}(b) \implies W(q) \approx \texttt{void} * \texttt{WILD}\}\\
&\quad \cup \ \{\mathit{Wild}(p, 0) \mid p \in \mathbb{N}\}.
\end{aligned}$$

The set $\|\alpha\|_W$ formalizes the invariants for CCured pointers and gives information about the types to which those pointers point.

First, a pointer of type $\tau * \texttt{SAFE}$ is either `null` or a valid pointer to a contiguous sequence of atomic values that conform to the type τ.

Second, a pointer of type $\tau * \texttt{SEQ}$ is either an integer disguised as a pointer, in which case both the b and e fields are null, or a valid `SEQ` pointer that may be out of bounds. In the simple case when $p = b$, the invariant is equivalent to the expected $\forall i.\ b \le p + i < e \implies W(p+i) \approx Nth(\tau, i \bmod \texttt{sizeof}(\tau))$. The modular arithmetic represents the fact that a $\tau * \texttt{SEQ}$ pointer usually points to a contiguous sequence of τs. The actual invariant $\forall q.\ b \le q < e \implies W(q) \approx$

$\mathit{Nth}(\tau, (q - p) \bmod \texttt{sizeof}(\tau))$ covers the possibility that $(p - b \bmod \texttt{sizeof}(\tau))$ may not be zero. For example, consider an $(\texttt{int} :: \texttt{int} :: \mathit{nil}) * \texttt{SEQ}$ pointer p. It might be cast to an $(\texttt{int} :: \mathit{nil}) * \texttt{SEQ}$ pointer using the rules described in Section 3.1, incremented by one word via pointer arithmetic, and then cast back to its original type. The new value $p+1$ is not aligned with respect to 2, the size of the original base type. However, the condition ensures that $W(p + 1) = \texttt{int}$ and $W(p+2) = \texttt{int}$, provided that both addresses are in bounds. Thus, we do not have to check SEQ pointer alignment at run-time: as long as the pointer is within bounds, dereferencing it yields a valid value of the base type. In cases where casts between arrays with varying elements would lead to a violation of this property, the typing premises in Figure 8 require that both pointers be WILD.

Finally, a WILD pointer is either an integer disguised as a pointer or a valid pointer with a base pointer. The base pointer is kept under system control and comes equipped with a mechanism for finding the length of the referenced area. Every word in the memory region specified by the base field is a valid WILD pointer (or disguised integer). This invariant holds regardless of where the user-controlled part of the original WILD pointer points. For the purposes of the proof, we assume that all WILD pointers are cast to $\texttt{void} * \texttt{WILD}$ before being written to memory. When an integer is read from an $\texttt{int} * \texttt{WILD}$ pointer, we instead read a $\texttt{void} * \texttt{WILD}$ value and cast it to an int. In practice, CCured uses tag bits to distinguish integers from pointers in such WILD areas.

We extend the notation $v \in \|\alpha\|_W$ element-wise to the corresponding notation for environments $\Sigma \in \|\Gamma\|_W$ (meaning $\forall x \in \mathit{Dom}(\Sigma).\ \Sigma(x) \in \|\Gamma(x)\|_W$). Recall that we assume all variables have atomic types (although they may point into larger structures).

At all times during the execution, the contents of each memory address must correspond to the typing constraints on that memory address. We say that such a memory is well-formed (written $\mathit{WF}_W(M)$), a property defined as follows:

$$\mathit{WF}_W(M) \overset{\text{def}}{\iff} \forall p \in \mathit{Dom}(M).\ M(p) \in \|W(p)\|.$$

There are several reasons why the evaluation of an expression can fail. The most obvious reason is that a CCured-inserted run-time check can fail. We actually consider this behavior to be safe. Another reason is that the operations on memory are undefined if they involve invalid addresses.

In order to state a progress theorem, we want to distinguish between executions that stop because memory safety is violated (i.e., trying to access an invalid memory location) and executions that stop because of a failed run-time check (an assert statement in rules of Figure 3 or Figure 4). We handle assertion failure by introducing a new possible outcome of evaluation. We say that $\Sigma, M \vdash e \Downarrow \mathit{CheckFailed}, M'$ when one of the run-time checks fails during the evaluation of the expression e. Technically, this approach requires that we add derivation rules that initiate the *CheckFailed* result when one of the run-time check fails and also rules that propagate the *CheckFailed* outcome from the subexpressions to the enclosing expression. We state below a theorem saying essentially that CCured programs never fail because of invalid memory accesses.

THEOREM 1 (PROGRESS). *If $\Gamma \vdash e : \tau$ and $\Sigma \in \|\Gamma\|_W$ and $WF_W(M)$ then either $\Sigma, M \vdash e \Downarrow CheckFailed, M'$ or else $\Sigma, M \vdash e \Downarrow v, M'$ and $v \in \|\tau\|_W$ and $WF_W(M')$.*

The proof of this theorem is a fairly straightforward induction on the structure of the typing derivations. This proof uses the invariant that memory is well-formed after every expression and command. From a safety perspective, the important aspect is the well-typed memory invariant. It not only captures the dynamic behavior of CCured pointers (e.g., a `SAFE` pointer is always either `null` or a valid pointer) but also higher-level soundness conditions (e.g., `WILD` pointers can never point to statically-typed pointers).

Next, we highlight a few corner cases in the proof that are particularly worth noting.

(1) *Memory reads*. In general, the run-time checks inserted by CCured (see Figure 3) combine with the $WF_W(M)$ invariants to ensure that valid and well-typed values are read. For example, when reading from a τ * `SAFE` pointer, the run-time check ensures that the pointer is not `null`. If it is not, the invariants ensure that $W(p+i) = Nth(\tau, i)$ for all $0 \leq i < \texttt{sizeof}(\tau)$: that is, that the value conforms to the expected type.

(2) *Memory writes*. Memory writes use all of the same checks as memory reads, so the program can only write through valid and in-bounds pointers. By induction, the value being written adheres to the invariants, so after the write, $WF_W(M')$ will hold for the new memory M', which differs from M only in locations p through $p + \texttt{sizeof}(\tau)$. The type mapping W does not change.

(3) *Pointer arithmetic*. Since the CCured type system does not allow pointer arithmetic on `SAFE` pointers, only the `SEQ` and `WILD` cases are interesting. It is important to note that only the p portion of a `SEQ` or `WILD` pointer value is changed by pointer arithmetic and that the bounds and length information remain under system control. A τ * `SEQ` pointer will always be incremented by multiples of $\texttt{sizeof}(\tau)$, so the invariant about the old pointer that held at positions modulo $\texttt{sizeof}(\tau)$ is sufficient to prove the invariant for the new pointer. `SEQ` and `WILD` pointers may stray out of bounds without violating our invariants, since run-time assertions will check the bounds when the pointer is dereferenced.

(4) *Casts from τ * `SEQ` to τ * `SAFE`*. CCured's run-time checks for casts require that the pointer $SEQ(p, b, e)$ be in bounds: $b \neq \texttt{null} \wedge b \leq p \leq e - \texttt{sizeof}(\tau)$. The invariant from $WF_W(M)$ on the old `SEQ` pointer is strong enough to prove the required invariant on the new `SAFE` pointer: take $q = p$.

(5) *Casts from τ_1 * `SEQ` to τ_2 * `SEQ`*. CCured permits such a cast provided that $\tau_1[n_1] \approx \tau_2[n_2]$ for some n_1 and n_2. We prove that, in this case, for any number i, $Nth(\tau_1, i \bmod \texttt{sizeof}(\tau_1)) \approx Nth(\tau_2, i \bmod \texttt{sizeof}(\tau_2))$, and that the invariant of the `SEQ` pointer is thus preserved.

In this case, there exists a type τ_0 (a prefix of both τ_1 and τ_2) of length $\gcd(\texttt{sizeof}(\tau_1), \texttt{sizeof}(\tau_2))$, and there exist also n'_1 and n'_2 such that $\tau_1 \approx$

$\tau_0[n'_1]$ and $\tau_2 \approx \tau_0[n'_2]$. In the common case where $n_1 = 1$ or $n_2 = 1$, τ_0 is the smaller one of τ_1 or τ_2.

We prove that $Nth(\tau_1, i \bmod \mathtt{sizeof}(\tau_1)) \approx Nth(\tau_0, i \bmod \mathtt{sizeof}(\tau_0))$ for any i. From $\tau_1 \approx \tau_0[n'_1]$ we know that $i \bmod \mathtt{sizeof}(\tau_1) = i \bmod \mathtt{sizeof}(\tau_0) + k_1 * \mathtt{sizeof}(\tau_0)$ for some positive k_1 less than n'_1. Thus $Nth(\tau_1, i \bmod \mathtt{sizeof}(\tau_1)) = Nth(\tau_0[n'_1], i \bmod \mathtt{sizeof}(\tau_0) + k_1 * \mathtt{sizeof}(\tau_0)) = Nth(\tau_0, i \bmod \mathtt{sizeof}(\tau_0))$. We prove a similar equality for τ_2 and obtain the desired result by transitivity.

Finally, note that the progress theorems state more than just memory safety. They also imply that well-typed computations of nondynamic type are type preserving (modulo subtyping), similar to corresponding results for a type-safe language. This result means that, for example, if a program reads through a $\tau *$SAFE pointer, the last value written there will have been a subtype of τ. Thus, CCured is memory safe and is also type safe for the nondynamic fragment.

5. HANDLING OTHER C FEATURES

The CCured language as formalized in Figures 3 and 4 includes pointers (reads and writes), pointer arithmetic, and type casts. This section explains how we handle some of the other potentially troublesome features of C.

5.1 Unions

The use of union types can lead to a violation of the type system because it is possible to write a value of one type into one field and then read the same value out of another field as a different type. Most implementations do not ensure that the field being read is the last one that was written. CCured offers three ways to guarantee type safety for unions:

(1) If all of the fields in the union have equivalent types, or if one field is a physical subtype of all of the others, then the union may be safely used without modification. For example, consider this union:

```
union circle_union {
  struct Figure f;
  struct Circle c;
};
```

`Circle` is a subtype of `Figure`, so `union circle_union` will have the same physical layout as `struct Circle`, and the `f` field simply allows access to a prefix of that structure. In our larger experiments, at least 50% of the unions fit this form.

(2) Unions may be annotated as `TAGGED`, in which case CCured will add a one-word tag to the representation of the union and then insert run-time checks to guarantee that the program only reads from the field to which it last wrote. In order for this approach to be safe, however, CCured will not allow programs to take the address of fields. (If it were possible to take the address of a field, the program could later read from that pointer even after a different type was stored in the union.) This tagging information often

duplicates a similar tag maintained by the programmer, but in general verifying that programs correctly implement and check their tags requires a dependent type system.

(3) The programmer may change, or instruct CCured to change, the union type to a struct. At the cost of additional space, this solution allows type safety even if the program takes the address of a field. Unfortunately, this approach also prevents CCured from catching errors that result from writing to field `a` and then reading from field `b`. Instead, the program will silently read whatever value was last written to `b`.

If none of these three strategies is used for a union, then any pointers involved will become dynamically typed, and the `WILD` pointer kind may spread throughout the program.

5.2 Variable-Argument Functions

Variable-argument functions in C are potentially unsafe since there is no language-level mechanism to ensure that the actual arguments agree in type and number with the parameters that the function uses.

CCured augments variable-argument functions to accept an additional parameter that specifies the number and types of the actual arguments. It then inserts run-time checks to verify, as each argument is processed, that the actual argument type matches what the function expects it to be. Finally, CCured modifies the call sites to variable-argument functions to pass the extra argument. The surprising aspect of the implementation is that this mechanism works without requiring any modification whatsoever to most variable-argument functions. (The primary exception is `printf`-like functions, which require annotations to indicate which argument contains the format string. In this case, CCured can take advantage of annotations already provided for the use of compilers such as `gcc`.)

However, this general approach does not work if the implementation of the variable-argument function is not instrumented by CCured; the `printf` library function is an obvious example. For external functions whose calling convention is known, we provide wrappers that check the argument correspondence before calling into the library. As a special case, if the format string is a literal at the call site, the correspondence can be checked statically and the wrapper bypassed.

5.3 Function Pointers

Because they are not subject to deallocation or arithmetic, function pointers are, in some respects, actually easier to handle than ordinary pointers. Function pointers that are not involved in casts are marked `SAFE`, and indirect calls through such pointers are handled the same way as direct calls (according to the types of the parameters).

However, if a function pointer is cast, either to another function type or to a data pointer type, the pointer is marked `WILD`. Furthermore, the types of all parameters to this function become `WILD`. Whereas `WILD` data pointers carry a pointer to their home area, WILD function pointers carry a pointer to a static descriptor of the function, which contains the function's true address and the number of expected arguments. (Descriptors are distinguishable from a regular dynamically typed area.) When we invoke a WILD function pointer, we check the descriptor to verify that we are passing the correct number of arguments and that the pointer value is the same as it was when first created (to detect arithmetic).

Unfortunately, function pointers are a common area of sloppiness in C, probably due to the complexity of their notation. Programmers routinely leave the parameters out of function pointer declarations, forcing CCured to mark these types WILD. In most of the experiments we discuss in Section 8, we chose to fix manually the declared types of these pointers to reduce the number of WILD pointers needed.

5.4 Heap Allocation

Some memory errors in C programs are due to dereferencing a heap pointer after the memory to which it refers has already been deallocated. CCured prevents this problem by replacing malloc with the Boehm-Demers-Weiser conservative garbage collector [Boehm and Weiser 1988], and making free do nothing. Since CCured maintains base pointers to valid memory objects, it is not possible to disguise a pointer in such a way that the collector would mistakenly believe a live object is unreachable. It is worth noting that, because CCured maintains enough information to distinguish pointers from integers, it would be possible to use a precise garbage collector.

In many cases, C programs include custom allocators built on top of malloc and free. Reasoning about the correctness of allocation and deallocation is very difficult, so CCured cannot prove the safety of such allocators directly. Instead, CCured requires the programmer to annotate custom allocators; based on these annotations, CCured will trust the cast at the allocation site, and it will initialize any allocated metadata as appropriate. Alternatively, the programmer can replace custom allocators with calls to malloc and free, which are handled as described above.

5.5 Stack Allocation

In C, it is possible to take the address of stack-allocated variables and to use the resulting pointers freely. This feature can potentially lead to memory errors if such pointers are used after the stack frame to which they point is deallocated at function return.

Since stack pointers are most commonly used to implement call-by-reference, which does not jeopardize memory safety, CCured enforces a restrictive policy that allows little else: stack pointers cannot be written into the heap or the globals, and they can only be stored in stack frames guaranteed to be deallocated before the frame to which they point. A key element of the enforcement of this policy is that stack pointers (or their home areas, for kinds that allow arithmetic) can be recognized by comparing their value with known bounds on the stack.

Programs that use stack pointers in a way that is inconsistent with the above policy must be modified to instead allocate the storage in question on the heap. CCured will make the appropriate change automatically for variables marked by the programmer with a special attribute called *heapify*. However, the cost of lost locality can be significant; the Spec95 benchmark li (Lisp interpreter) slows down by about 25% due to heapified variables.

5.6 Sizeof

The sizeof operator reports the size of a given type's representation. By itself, this feature does not present a problem; CCured makes data structures bigger, but the values produced by sizeof increase accordingly. However, two types that were the same before transformation by CCured may become unequal afterward, depending on their involvement in the inference algorithm, and this scenario can lead to problems.

The most common example arises from memory allocation. For example, a program that allocates an array of 5 pointers might contain the statement:

```
int **p = (int**)malloc(5 * sizeof(int*));
```

If the program actually uses any of these pointers beyond the first one, p will be inferred at least SEQ. But the int* type in the sizeof expression will not be connected to any other type in the inference algorithm, and consequently, it will be inferred SAFE. Hence, the call to malloc will only allocate space for 5 SAFE pointers, and the program will abort with an array bounds violation when it tries to access the second or third array element.

To fix this problem, the statement could be changed to

```
int **p = (int**)malloc(5 * sizeof(*p));
```

Now, the argument to sizeof is connected with the type of the result. In general, any sizeof(*type*) expression is suspect if *type* contains pointers that are not enclosed in a struct or union definition. CCured will identify such expressions and request that the programmer change them to the appropriate sizeof(*expression*).

Fortunately, only a small fraction of sizeof expressions require this modification. The rest have forms such as

```
int* p = (int*)malloc(5 * sizeof(int));
struct T* q = (struct T*)malloc(5 * sizeof(struct T));
```

There are no disconnected pointers here, since any pointers inside struct T will share the same qualifiers as all other references to struct T. Of the 650 uses of sizeof in OpenSSH, CCured identified just 12 locations requiring user intervention; these results are typical of our experiments.

5.7 Structure Padding and Alignment

The type equality relation described so far (Section 1) assumes that atomic elements of an aggregate type (structure or array) are laid out in memory contiguously. In this section we extend type equality (and hence subtyping) to take into account the padding typically inserted by compilers to improve performance and/or adhere to alignment requirements imposed by the underlying architecture.

Ideally, we could base our formalization of padding on the ANSI C standard [ISO/IEC 1999]. However, the standard actually gives implementations a lot of freedom in how to do padding, so much so that some important instances of physical subtyping are technically not portable. However, since popular implementations such as the GNU and Microsoft C compilers use relatively predictable padding strategies, a fact used by C programs in practice, our formalism is designed to model the commonly expected behavior.

As will be demonstrated shortly, we cannot simply compute the structure padding in advance before qualifier inference, since the representation changes implied by the choice of qualifiers can affect the padding ultimately inserted. So instead, we model the padding behavior by inserting variable-size "padding elements" into the sequence of atomic types. The width of padding element "pad(n)" is in $[0, n-1]$ such that it ends on an address that is a multiple of n, which must be a power of 2 and is typically 4 or 8.

We use the following algorithm to insert padding elements:

- All data elements are preceded by pad(n), where n is the width of that element.
- All structures begin and end with pad(n), where pad(n) is the largest padding element that appears anywhere in the structure.

Furthermore, to eliminate what would otherwise be spurious mismatches due to extra layers of structs, we rewrite the sequence into a canonical form:

- pad(n) followed by pad(m) is collapsed to pad(max(n, m)).
- An element α followed by pad(n) is collapsed to α, if the width m of α is at least n, since it will have been preceded by pad(m) as well.

For example, gcc's behavior on x86 with the `-malign-double` flag can be modeled by adding a pad(8) element before every double and at the beginning and end of every structure that (transitively) contains a double:

$$\begin{array}{lcrll}
\texttt{struct T \{} & & & \mathrm{pad}(8) & :: \\
\quad\texttt{struct S \{} & & & \mathrm{pad}(8) & :: \\
\qquad\texttt{double d;} & \phi & & \texttt{double} & :: \\
\qquad\texttt{int x;} & \longrightarrow & & \texttt{int} & :: \\
\qquad\texttt{int} * q_1 \texttt{ y;} & & & \texttt{int} * q_1 & :: \\
\quad\texttt{\} s;} & & (*) & \mathrm{pad}(8) & :: \\
\quad\texttt{int} * q_2 \texttt{ i;} & & & \texttt{int} * q_2 & :: \\
\quad\texttt{int j;} & & & \texttt{int} & :: \\
\texttt{\};} & & & \mathrm{pad}(8) & :: \mathit{nil}
\end{array}$$

This example demonstrates why it is not sufficient to simply compute the padding when CCured first reads the structure. In the example, if q_1 is `SAFE`, then all padding elements have width 0, but if q_1 is `WILD`, then the padding element marked (*) has width 4. Had we prematurely computed the padding under the assumption that q_1 would be SAFE, we would have believed that T is compatible with a flattened version that lacked the (*) element. This is why we keep track of *potential* padding, not actual padding, since we can't know the latter until after qualifier inference has finished.

The crucial fact about padding layouts is that a given sequence of atomic types, including padding elements, has a fixed layout in memory if it is preceded by a padding element pad(N) where $N \geq n$ for any pad(n) in the sequence. Given two starting addresses x and y, both congruent to 0 (mod N), we show by induction on the length of the sequence A of atomic types that the widths B_x and B_y of every prefix B of A are equal in the memory layouts beginning at x and y, respectively. For the empty initial prefix ϵ, ϵ_x and ϵ_y are both 0 and therefore equal. For the sequence prefix $B\alpha$, $B_x = B_y$ by the inductive hypothesis. Therefore the addresses of α, $x + B_x$ and $y + B_y$ are congruent (mod N). If $\alpha = \text{pad}(n)$, then $x + B_x = y + B_y \pmod{n}$ as well since n divides N, and therefore the width of α will be the same for both sequences: $B\alpha_x = B\alpha_y$. If α is not a padding element then it has a fixed size, and again $B\alpha_x = B\alpha_y$.

Given the compatibility of equal layout sequences, we extend type compatibility ($\approx$) for atomic types so that a given padding element is equal only to itself. This constraint ensures that two types that are compatible will be laid out such that all atomic constituents appear at the same offsets, and will therefore be accessed with consistent typing assumptions.

6. COMPATIBILITY WITH LIBRARIES

It is often necessary to link transformed programs with *external* libraries that were not compiled by CCured. Doing so allows users to avoid recompiling these libraries with each program. More importantly, this feature allows CCured's output to be linked with binaries written in assembly code or other languages, and it allows programmers to use libraries for which the source code is unavailable.

CCured will mangle the name of any function or function declaration whose type transitively contains a non-SAFE pointer. As a result, if a program attempts to pass a wide pointer to an external library function or receive a wide pointer in return, the program will fail to link rather than cause a run-time error. When programmers need to link with an external function that makes nontrivial use of pointers, they must tell CCured how to handle this problem using one of the mechanisms described in this section.

CCured likewise mangles the names of any global variable that uses a non-SAFE pointer. However, using global pointers to exchange data across a library boundary is much less common than calling a library function. The easiest way to use such a variable with CCured is to replace any references to it with *getter* and *setter* functions, and then use one of our mechanisms that deal with functions.

6.1 Library Wrappers

One approach to the problem of library compatibility is to write wrapper functions for external library functions. To link correctly with a function that is not instrumented, CCured must do the following:

```
#pragma ccuredwrapper("strchr_wrapper", of("strchr"))
char* strchr_wrapper(char* str, int chr) {
  __verify_nul(str);                         // check for NUL termination.
  char *result = strchr(__ptrof(str), chr);  // call underlying function, stripping metadata from str
  // the return value will point into str's home area, so build a multi-word
  // pointer for the return value using str's metadata.
  return __mkptr(result, str);
}
```

Fig. 10. A wrapper for strchr.

(1) Determine what constraints the external function places on its inputs to ensure safe operation. Although there is no way to guarantee that the external function is memory-safe, CCured can validate assumptions on which the function relies, such as the size of an input buffer.

(2) Perform appropriate run-time actions to check that these constraints are met and to pack or unpack multiword pointers.

We accomplish these tasks by requiring the programmer to provide a small wrapper specification for each external function that has a mangled name. For example, Figure 10 shows a wrapper specification for strchr, a function that returns a pointer to the first occurrence of a given character in a string. CCured replaces every call to strchr with a call to this wrapper, which has the same signature as strchr. We provide a set of helper functions such as __verify_nul, __ptrof, and __mkptr that are replaced with specialized code depending on the pointer kinds of their arguments and results. For example, when str is a WILD pointer, __ptrof(str) performs a bounds check, verifies that the tag bits are correct, and returns str.p. CCured analyzes wrapper functions in a context-sensitive way, so a single wrapper function can work with any set of inferred qualifiers. Notice that the wrapper specification can also include checks of other preconditions of the library function beyond those needed for memory safety.

We have implemented wrappers for about 120 commonly used functions from the C Standard Library. The wrappers are packaged with CCured so that calls to these functions are correctly handled with no further intervention required.

6.2 Compatible Metadata Representations

The wrapper specifications described above have proved useful for relatively simple external functions such as the core of C's standard library. However, C programs often make use of library interfaces that are much more complex. Consider, for example, the library function gethostbyname(), which is used to perform DNS queries in some of the network servers on which we want to use CCured. This function returns a pointer to the following structure (omitting some fields for clarity):

```
struct hostent {
  char *h_name;         // String
  char **h_aliases;     // Array of strings
  int h_addrtype;
};
```

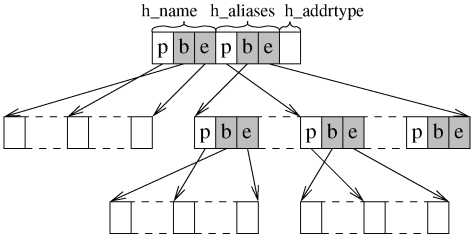


Fig. 11. Interleaved representation for `struct hostent`. Array-bounds metadata (gray) is interspersed with data (white).

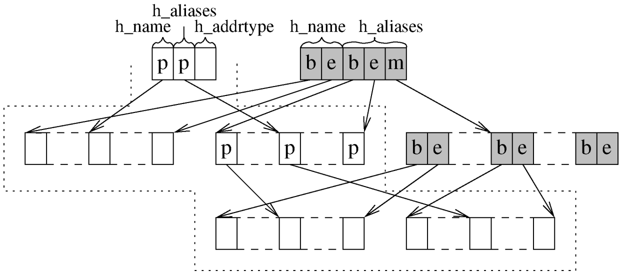


Fig. 12. Noninterleaved representation for `struct hostent`. Metadata (gray) has been separated into a parallel data structure so that the data (white, boxed) has the format expected by the C library. The `m` field is a pointer to the array's metadata.

Since the library that creates this structure is not instrumented by CCured, it returns data in exactly this format. However, CCured needs to store metadata (`b` and `e` fields) with each string and with the array of strings itself; in other words, CCured expects a representation in which all pointers are wide pointers, as shown in Figure 11. In order to *convert* the library's data representation to CCured's data representation, we would have to do a deep copy of the entire data structure. Since deep copies require expensive allocations and destroy sharing, such a conversion is undesirable. Conversions can be avoided if the metadata is not interleaved with the normal data; however, merely moving the metadata to the beginning or the end of the structure is insufficient in a number of cases (e.g., an array of structures used by a library).

Our solution is to split the data and metadata into two separate structures with a similar shape; for example, a linked list is transformed into two parallel linked lists, one containing data and the other containing metadata. Creating and maintaining these data structures is quite easy. For every data value in the original program, our transformed program has a data value and a metadata value. Every operation on a value with metadata is split into two such operations, one on the data and one on the corresponding metadata. Figure 12 shows the representation of `struct hostent` using this new approach.

We will now make our informal notion of separated data and metadata more precise by specifying the types of these values. The data value's type must be identical to the original C type, since we intend to pass it directly to an external library (or obtain it directly from a library). For a given CCured type τ, we express this original C type (without pointer kinds) as $\mathsf{C}(\tau)$. Similarly,

$$\begin{array}{lcl}
\mathsf{C}(\texttt{void}) & = & \texttt{void} \\
\mathsf{C}(\texttt{int}) & = & \texttt{int} \\
\mathsf{C}(\mathrm{struct}\{\ \ldots \tau_i\ f_i; \ldots\ \}) & = & \mathrm{struct}\{\ \ldots \mathsf{C}(\tau_i)\ f_i; \ldots\ \} \\
\mathsf{C}(\tau * \texttt{SAFE}) & = & \mathsf{C}(\tau) * \\
\mathsf{C}(\tau * \texttt{SEQ}) & = & \mathsf{C}(\tau) * \\
\mathsf{Meta}(\texttt{void}) & = & \texttt{void} \\
\mathsf{Meta}(\texttt{int}) & = & \texttt{void} \\
\mathsf{Meta}(\mathrm{struct}\{\ \ldots \tau_i\ f_i; \ldots\ \}) & = & \mathrm{struct}\{\ \ldots \mathsf{Meta}(\tau_i)\ f_i; \ldots\ \} \\
\mathsf{Meta}(\tau * \texttt{SAFE}) & = & \mathrm{struct}\{\ \mathsf{Meta}(\tau) * m;\ \}^{\dagger} \\
\mathsf{Meta}(\tau * \texttt{SEQ}) & = & \mathrm{struct}\{\ \mathsf{C}(\tau) * b, * e; \\
 & & \qquad\quad \mathsf{Meta}(\tau) * m;\ \}^{\dagger}
\end{array}$$

† The m field is omitted if $\mathsf{Meta}(\tau) = \texttt{void}$.

Fig. 13. The C and Meta functions define the data and metadata types (respectively) in the compatible representation. Together, these functions define a compatible alternative to the Rep function.

we write the type of the separated metadata value as $\mathsf{Meta}(\tau)$. Together, the types $\mathsf{C}(\tau)$ and $\mathsf{Meta}(\tau)$ provide a complete representation for the CCured type τ; thus, they can be used in place of the representation given by $\mathsf{Rep}(\tau)$ in Figure 2.

Formal definitions for the functions C and Meta are given in Figure 13. The definition for C recursively strips off all pointer qualifiers; for example, $\mathsf{C}(\texttt{int} * \texttt{SEQ} * \texttt{SEQ}) = \texttt{int} **$. The definition of the function Meta is slightly more complex, but it adheres to the following rule of thumb: the metadata for a type τ must include the metadata required by τ itself (e.g., a SEQ pointer's b and e fields) as well as the metadata for any base types. Thus, the metadata for a SEQ pointer includes a base pointer, an end pointer, and a pointer to the metadata of its base type, if such metadata exists. A SAFE pointer has no metadata of its own, so it only needs to maintain a pointer to the metadata of its base type, if the base type requires metadata. Likewise, a structure requires no metadata in and of itself, so its metadata is simply a structure containing the metadata of each of its fields, as necessary.

An important property of the Meta function is that metadata is only introduced by pointers that have metadata in their original CCured representation as given by the Rep function (e.g., SEQ pointers); if a type does not contain any of these pointers, its metadata type will be void. On the other hand, any type that is composed from a pointer that needs metadata must itself have metadata, since at the bare minimum it must maintain a pointer to the component pointer's metadata. This case illustrates the disadvantage of using the separated metadata representation: pointers require more metadata than before, and in some cases, even SAFE pointers require metadata.

Because this new representation is less efficient than the original one, we restrict its use to those parts of a program that require it for compatibility. To indicate which representation should be used for a given type, we add two new type qualifiers: SPLIT and NOSPLIT. Note that unlike the SAFE and SEQ qualifiers, which apply only to pointer types, these new qualifiers apply to all types. A value of type τ SPLIT is represented using data of type $\mathsf{C}(\tau)$ and metadata of type $\mathsf{Meta}(\tau)$. Correspondingly, a value of type τ NOSPLIT is represented using the type $\mathsf{Rep}(\tau)$, which contains interleaved data and metadata.

$$\begin{array}{l}
\mathsf{Rep}(\texttt{struct}\{\ \ldots\tau_i\ \texttt{SPLIT}\ f_i;\ldots\ \}) = \\
\qquad \texttt{struct}\{\ \ldots\mathsf{C}(\tau_i)\ f_i;\ \mathsf{Meta}(\tau_i)\ m_i;\ldots\ \} \\
\mathsf{Rep}(\tau\ \texttt{SPLIT} * \texttt{SAFE}) = \\
\qquad \texttt{struct}\{\ \mathsf{C}(\tau) * p;\ \mathsf{Meta}(\tau) * m;\ \} \\
\mathsf{Rep}(\tau\ \texttt{SPLIT} * \texttt{SEQ}) = \\
\qquad \texttt{struct}\{\ \mathsf{C}(\tau) * p, * b, * e;\ \mathsf{Meta}(\tau) * m;\ \}
\end{array}$$

Fig. 14. The Rep function can be extended to handle NOSPLIT types that contain SPLIT types. This definition extends the definition given in Figure 2, which considers only NOSPLIT types.

SPLIT pointers cannot point to NOSPLIT types; otherwise, they would be incompatible with external libraries. However, NOSPLIT pointers are allowed to point to SPLIT types. The representation of such "boundary" pointers is given in Figure 14, which extends the previous definition of Rep to handle this case. For example, a SAFE NOSPLIT pointer to a SPLIT type τ consists of pointers to τ's data and metadata, which are represented using $\mathsf{C}(\tau)$ and $\mathsf{Meta}(\tau)$. SEQ pointers and structures are handled in a similar manner.

Example. The following example demonstrates the transformation applied when using CCured's compatible representation:

```
struct hostent SPLIT * SAFE SPLIT h1;
struct hostent SPLIT * SAFE NOSPLIT h2;
char * SEQ SPLIT * SEQ SPLIT a;
a = h1->h_aliases;
h2 = h1;
```

In this program, struct hostent uses the compatible representation as shown in Figure 12. We declare two pointers to this structure, one SPLIT and one NOSPLIT. We copy h1's h_aliases field into the local variable a of the same type, and then we assign the SPLIT pointer h1 to the NOSPLIT pointer h2. The instrumented program is as follows:

```
struct meta_seq_char { char * b, * e; };
struct meta_seq_seq_char {
    char * * b, * * e;
struct meta_seq_char *m;
};
struct hostent * h1;
struct meta_hostent {
    struct meta_seq_char h_name;
    struct meta_seq_seq_char h_aliases;
} * h1m;
struct {
    struct hostent * p;
    struct meta_hostent * m;
} h2;
char * * a;
struct meta_seq_seq_char am;
a = h1->h_aliases;     am = h1m->h_aliases;
h2.p = h1;             h2.m = h1m;
```

In the transformed program, the SPLIT pointers h1 and a are now represented using two pointers each: h1, h1m, a, and am. In these variable names, the "m" designates a metadata value. The type of h1m is as shown in the right-hand side of Figure 12. The NOSPLIT pointer to a SPLIT struct hostent is represented as a structure containing pointers to the data and metadata of the underlying SPLIT structure. The assignment to a becomes two assignments, one for the data part stored in a and one for the metadata part stored in am. Note that we dereference h1m in the same way that we dereference h1; the metadata structure is traversed in parallel with the data structure. The conversion from the SPLIT pointer h1 to the NOSPLIT pointer h2 simply copies the data and metadata pointers into h2.

6.2.1 *Changes to the Inference Algorithm.* CCured requires that the programmer identify places in the program where this compatible representation should be used. To assist the programmer, CCured provides an inference algorithm that spreads the SPLIT qualifier as far as necessary based on the programmer's annotations. This inference algorithm is implemented as a small extension (about 150 lines) to the existing CCured inference algorithm.

Initially, all types are assumed to be NOSPLIT unless the programmer indicates otherwise. Starting from user-supplied SPLIT annotations, SPLIT qualifiers flow down from a pointer to its base type and from a structure to its fields in order to ensure that SPLIT types never contain NOSPLIT types.

In addition, if there is a cast from a SPLIT type to a NOSPLIT type (or vice versa), we ensure that all types contained within the NOSPLIT type are SPLIT; that is, a cast between pointers requires that both base types be SPLIT, and a cast between structures requires that all fields be SPLIT. This restriction corresponds directly to a restriction in the pointer qualifier inference algorithm; in both cases, converting between pointer types whose base types have different representations is unsound. To obtain the type qualifiers in the above example, the programmer would only have to annotate the top-level type of h1 and a to be SPLIT (possibly because they are being passed to or from library functions). The remaining SPLIT and NOSPLIT qualifiers are then inferred based on the rules we described above.

6.2.2 *Limitations.* This compatible metadata representation significantly eases the burden of communicating with external libraries, but unfortunately, it does not solve the entire problem. In particular, if a library makes changes to a data structure that require corresponding changes to the associated metadata, then the metadata will be invalid upon return from the external library. Also, CCured must generate new metadata when the library returns a newly allocated object. Thus, CCured must validate any new or potentially modified data structures after calling into an external library function. We are currently evaluating a number of strategies for coping with this problem. However, experience suggests that this compatible representation is useful even in the absence of such mechanisms. Many data structures are read-only for either the application or the library, which simplifies or eliminates this problem; for instance, applications rarely modify the struct hostent returned by gethostbyname(), which simplifies the problem of generating metadata for its return value. In other cases, such as the function `recvmsg()`, the library only modifies a character buffer that has no associated metadata.

We currently do not support this compatible representation for `WILD` pointers, in large part due to the limitations we have discussed in this section. Since `WILD` pointers maintain metadata for every word in memory regardless of its static type, we must update the `WILD` pointer's metadata to account for *every* change made by the external library, not just changes to pointers. This added complexity makes it far more difficult to overcome the limitations discussed above. Furthermore, we usually attempt to eliminate `WILD` pointers from transformed programs for efficiency reasons alone. Thus, even though it is quite possible to implement compatible `WILD` pointers using the techniques discussed in this section, we prefer to replace `WILD` pointers with `SAFE`, `SEQ`, or `RTTI` pointers at library boundaries.

7. CURING A PROGRAM

We have applied the techniques described in this article to a number of real-world programs ranging from several thousand lines to over 300,000 lines. This section provides an overview of the process of "curing" a program (that is, the process of using CCured to ensure the memory safety of a program). Included with each step is a rough estimate of the approximate amount of time required for a 100,000-line program.

(1) *Adjust build process.* The first step when curing a program is to modify the build process to use CCured instead of the default compiler. Using the CIL infrastructure [Necula et al. 2002b], CCured will merge all source code for the program and its internal libraries into a single file in preparation for CCured's analysis. Adjusting the build process to use CCured typically takes less than 1 hour.

(2) *Annotate variable argument functions and sizeof.* CCured's standards for C code are somewhat stricter than those of a typical compiler. For example, CCured expects annotations on some `printf`-like functions (Section 5.2). Also, in some cases, CCured requires the programmer to change `sizeof`(τ) to `sizeof`(e) where e is an expression having type τ (Section 5.6). CCured emits warnings in all of these cases. Large programs typically have about a dozen such cases, which are easily fixed by hand. As a concrete example, `bind` is more than 300,000 lines of code and has exactly 10 such `sizeofs` spread over four source files. This step is also typically straightforward and takes less than 1 hour.

(3) *Eliminate bad casts.* Once CCured's warnings have been addressed, the inference algorithm will identify bad casts. Without help from the programmer, CCured will typically be unable to prove the safety of several hundred casts in a 100,000-line program; CCured will treat all pointers involved in such casts as `WILD`.

At this point, it is entirely possible to move to the next step. However, `WILD` pointers require more space and time overhead than other CCured pointers (Section 3), and they complicate the process of linking with external libraries (Section 6). Thus, we typically choose to invest programmer effort into eliminating bad casts by annotating or modifying the program. Because a single bad cast can cause a large percentage of the pointers in a program to become `WILD`, eliminating bad casts is typically an all-or-nothing proposition.

Several approaches are useful here:

—*Annotating allocators and context-sensitive functions.* Custom allocators require a trusted cast, because otherwise CCured would have to reason about correct allocation and deallocation of memory. By marking a custom allocator as a trusted allocator, the programmer can eliminate all bad casts caused by CCured's inability to reason about allocation. Alternatively, custom allocators can be replaced by calls to `malloc` and `free`, which will use a garbage collector. (See Section 5.4 for details.) If a function is simply a wrapper around `malloc`, it should be declared polymorphic so that CCured knows that the type of the return value is context-sensitive.

—*Run-time type information and physical subtyping.* CCured can identify potential upcasts and downcasts. The programmer can then confirm these hypotheses by inserting annotations identifying structures that are subtypes of one another and by inserting "seed" `RTTI` pointers that are starting points for the `RTTI` inference algorithm (Section 3.2). We decided not to infer `RTTI` pointers completely automatically because the programmer often needs to verify that the `RTTI` mechanism correctly captures the program's behavior. For instance, the programmer must ensure that an appropriate upcast precedes every downcast so that `RTTI` information will be sufficiently precise at the point of the downcast; otherwise, a CCured run-time check will fail.

—*Rewriting code.* Often, a slight modification to the original code can allow CCured to prove the safety of code that it could not previously understand. The most common case is removing extraneous incorrect casts, such as casting to `char*` before calling `free`. Another common case is function pointer types that omit argument declarations (Section 5.3).

—*Trusted casts.* As a last resort, a programmer can hide an unwanted bad cast by telling CCured to trust the cast.

Using the above mechanisms, the programmer should be able to eliminate all of the bad casts in the program. This is typically the most time-consumig part of the curing process. We estimate that this could take 20 or more hours for a 100,000-line program. If the program contains many "dirty" tricks this step can take significantly more than that.

(4) *Fix linker errors.* After inferring pointer kinds, CCured will compile the program and attempt to link it with the appropriate external libraries. Depending on the libraries used, the cured program may fail to link due to the name mangling mechanism described at the beginning of Section 6; such an error indicates that CCured needs more information in order to link properly with external libraries. There are several possible solutions to such problems:

—*Compatible pointers.* The programmer can mark external interfaces that should use CCured's compatible representation (Section 6.2). The programmer must also identify "out" parameters—that is, parameters that may return newly allocated data structures. In doing so, the programmer must verify that pointers in all other structures are treated as read-only by the external library.

—*Wrapper functions.* A second alternative is to write a wrapper that allows CCured to interact with the external library (Section 6.1). This approach requires more effort on the part of the programmer, but it also gives the programmer more control over CCured's library interface. This method is especially useful when many cured programs will be linked against the same library, since the cost of writing wrappers is amortized.

—*Cure the library.* A final solution is to incorporate library code into the merged program, which allows CCured to cure the library in addition to the program itself. In some cases, this approach can simplify the task of linking with the external library. However, in other cases, when the library is very low-level or when duplicating library code in each executable is undesirable, one of the other two approaches is preferable.

With these techniques, the programmer can ensure that the cured program correctly links with external libraries. This step takes time proportional to the number of new wrappers to be written. Once we have collected wrappers for the most frequently used library functions we have found that we can finish this step in about 4 hours.

(5) *Test and debug.* Finally, the programmer needs to run suitable regression tests to verify that the cured program functions as expected. CCured's run-time checks can fail in some cases where the program is actually memory-safe. A frequent cause of such errors is pointers from the heap to the stack; the programmer can fix these errors by adding "heapify" annotations, which instruct CCured to store the appropriate data on the heap instead of on the stack (Section 5.5). In other cases, the failure of a CCured run-time check indicates the presence of an actual memory safety violation. In our experiments we have been extra careful to not change the semantics of the code when making the changes necessary to cure it. Consequently, it typically took less than 8 hours to perform this step.

7.1 Limitations

CCured will not work with some shared-memory multithreaded programs, since programmers may assume that pointers occupy a single machine word, meaning writes of pointer values are atomic. Multiple instructions are needed to write fat pointers to memory, and the pointer may be in an inconsistent state between these instructions. We believe CCured is safe for programs that acquire locks on shared memory before accessing it.

Interfacing with external code is difficult in the presence of `WILD` pointers. Eliminating bad casts, and hence `WILD` pointers, is the most time-consuming step of porting a C program to CCured, and library compatibility is the major

Name	Lines of Code	% sf/sq/w/rt	CCured Ratio	Purify Ratio	Memory Ratio	Lines Changed	Trusted Casts
SPECINT95							
compress	1590	90/10/0/0	1.17	28	1.01	36	0
go	29,315	94/06/0/0	1.22	51	1.60	117	0
ijpeg	31,371	80/20/0/1	1.50	30	1.05	1103	0
li	7761	80/20/0/0	1.70	50	2.00	600	0
Olden							
bh	2053	80/20/0/0	1.44	94	1.55	271	0
bisort	707	93/07/0/0	1.09	42	2.00	469	0
em3d	557	93/06/0/0	1.45	7	1.39	22	0
health	725	93/07/0/0	1.07	25	1.90	449	0
mst	617	97/03/0/0	1.87	5	1.15	44	0
perimeter	395	100/0/0/0	1.10	544	1.97	3	0
power	763	94/06/0/0	1.29	53	1.58	8	0
treeadd	385	96/04/0/0	1.15	500	2.61	14	0
tsp	561	100/0/0/0	1.06	66	2.54	7	0
Ptrdist-1.1							
anagram	661	88/12/0/0	1.43	34	1.52	37	0
bc	7323	77/23/0/0	9.91	100	2.18	58	0
ft	2194	98/02/0/0	1.03	12	2.12	59	0
ks	793	88/12/0/0	1.11	31	1.65	22	0
yacr2	3999	88/12/0/0	1.56	26	1.63	30	0

Fig. 15. CCured performance on three benchmark suites. The measurements are presented as ratios, where 1.50 means the program takes 50% longer to to run when instrumented with CCured. The "sf/sq/w/rt" column shows the percentage of static pointer declarations that were inferred SAFE, SEQ, WILD, and RTTI, respectively. The "Memory ratio" column measures the overhead in terms of greatest total virtual memory size (as reported by the operating system).

reason that WILD pointers are undesirable. Additional support for adding WILD metadata to buffers returned by a library may help here.

8. EXPERIMENTS

We tested CCured on many real-world C programs ranging in size from several hundred lines of code to several hundred thousand. These experiments allowed us to measure both the performance cost of the run-time checks inserted by CCured and the amount of manual intervention required to make existing C programs work with our system. In general, computationally expensive tasks like the Spec95 benchmarks and the OpenSSL cryptography library showed the greatest slowdown (ranging from 0–87% overhead). System software like Linux kernel modules and FTP daemons showed no noticeable performance penalty; the cost of run-time checks is dwarfed by the costs of input/output operations. Our experiments allowed us to detect a number of bugs in existing programs and enabled us to run safety-critical code without fear of memory-based security errors such as buffer overruns.

Figure 15 shows the results of using CCured with the Spec95 [SPEC 1995], Olden [Carlisle 1996], and Ptrdist-1.1 [Austin et al. 1994] benchmark suites. Using CCured required minor changes to some of these programs, such as correcting function prototypes, trusting a custom allocator, or moving to the heap some local variables whose address is itself stored into the heap. These changes

Name	funptr	sarray	heap	narray	rtti	vararg	calloc	union	scanf	csfun	sizeof
compress	X	X									
go				X							
ijpeg	X				X			X			X
li			X					X			X
bh						X	X	X			
yacr2				X					X		X
gzip	X		X							X	
ftpd		X				X		X			
OpenSSL	X		X	X	X	X	X	X		X	X
OpenSSH			X			X	X		X	X	X
sendmail	X		X		X	X	X	X		X	X
bind	X				X	X		X	X	X	X

Fig. 16. This table indicates CCured features, manual annotation burdens, and general notes about selected benchmark programs. For example, a benchmark noted with calloc contained a custom allocator that was hand-annotated and treated as trusted by CCured. The symbols are defined as follows. funptr: extensive use of function pointers (Section 5.3); sarray: buggy (overflowable) stack-allocated arrays; heap: stack-allocated variables were moved to the heap (Section 5.5); narray: nested arrays and nontrivial SEQ-SEQ casting (Section 3.1); rtti: RTTI pointers and an explicit class hierarchy (Section 3.2); vararg: custom variable-argument functions (Section 5.2); calloc: trusted custom allocator; union: unions not handled by method (1) in Section 5.1; scanf: complex use of scanf to create values; csfun: functions annotated as "context-sensitive"; sizeof: sizeof changed from type to expression (Section 5.6).

resulted in modifications to about 1 in 100 lines of source code. The benchmark and CCured features of particular interest for selected benchmarks are tabulated in Figure 16. In particular, this indicates that our performance on the larger benchmarks was not because they are particularly well-behaved. In the process we discovered a number of bugs in these benchmarks: ks passes a FILE* to printf where a char* is expected, compress and ijpeg contain array bounds violations, and go has eight array bounds violations and one use of an uninitialized variable as an array index. Most of the go bugs involve the use of multidimensional arrays, which demonstrates an important advantage of our type-based approach: if we viewed the entire multidimensional array as one large object, some of those bugs would not be detected.

For almost all the benchmarks, CCured's safety checks added between 3% and 87% to the running times of these tests. For comparison, we also tried these tests with Purify version 2001A [Hastings and Joyce 1991], which increased running times by factors of 5–100. Purify modifies C binaries directly to detect memory leaks and access violations by keeping 2 status bits per byte of allocated storage. Purify has an advantage over CCured in that it does not require the source code to a program (or any source code changes), so it is applicable in more situations. However, without the source code and the type information it contains, Purify cannot statically remove checks as CCured does. Also, Purify does not catch pointer arithmetic between two separate valid regions [Jones and Kelly 1997], a property that Patil and Fischer [1997] showed to be important.

The bc program takes 10 times as long to run after curing. Almost all of the overhead is due to using a garbage collector, as bc allocates many short-lived objects (it is a calculator interpreter). If the garbage collector is disabled, instead trusting the program's calls to `free()`, the overhead drops to less than 50%. See Hirzel [2000], p. 31, for a more detailed analysis of this program's behavior with a collector. Presumably, this program could be rewritten to be collector-friendly by keeping a cache of objects to reuse instead of making so many allocation requests.

CCured's program transformations incurred a memory overhead of 1–284%, as measured by comparing the great amount of virtual memory ever used by the process. This includes wider pointers, increased code size for run-time checks, and linking with CCured's run-time system. We also ran comparisons against Valgrind [Seward 2003], an open-source tool for finding memory-related bugs. Valgrind checks all reads, writes, and calls to allocation functions via JIT instrumentation, as well as maintaining 9 status bits per bit of program memory. Like Purify, it does not require the program source but entails a steep run-time overhead; Valgrind slows down instrumented programs by factors of 9–130, as shown in Figure 18. For `compress` and `treeadd`, the benchmarks for which CCured had the best and worst memory overhead, Valgrind used 3.05 and 11.35 times as much memory as the original program. Both Purify and Valgrind miss many memory errors that CCured catches; in particular, they do not catch out-of-bounds array indexing on stack-allocated arrays, which is an error often exploited by viruses when gaining control of a system.

8.1 Interacting with C Code

As we began to tackle larger programs that relied heavily on the C Standard Library and on other preexisting C binaries, we found that CCured had no convenient way to link with such code. Our first solution to this problem was the system of wrappers described in Section 6.1.

These wrappers helped us use CCured to make memory-safe versions of a number of `Apache` 1.2.9 modules. Buffer overruns and other security errors with Apache modules have led to a least one remote security exploit [SecuriTeam.com 2000]. In addition to writing CCured wrappers for Apache's array-handling functions, we annotated data structures that are created by Apache and passed to the module so that they would be inferred as having `SAFE` pointers. The physical subtyping described in Section 3.1 was necessary for CCured to determine that some casts were safe.

Figure 17 shows the performance of these modules on tests consisting of 1000 requests for files of sizes of 1, 10, and 100K. The `WebStone` test consists of 100 iterations of the WebStone 2.5 `manyfiles` benchmark with every request affected by the `expires`, `gzip`, `headers`, `urlcount`, and `usertrack` modules. In general, very little work was involved in getting a particular Apache module to work with CCured. For example, `gzip` required changes to 26 lines. This included annotating a `printf`-like variable-argument function and changing the declared type of a buffer from `void*` to `char*`. The change in speed due to CCured was well within the standard deviation in these measurements.

We also used CCured on two Linux kernel device drivers: `pcnet32`, a PCI Ethernet network driver, and `sbull`, a ramdisk block device driver. Both were

Module Name	Lines of Code	% sf/sq/w/rt	CCured Ratio	Lines Changed	Trusted Casts
`asis`	149	72/28/0/0	0.96	2	0
`expires`	525	77/23/0/0	1.00	5	0
`gzip`	11,648	85/15/0/0	0.94	136	0
`headers`	281	90/10/0/0	1.00	6	0
`info`	786	86/14/0/0	1.00	62	3
`layout`	309	82/18/0/0	1.01	37	0
`random`	131	85/15/0/0	0.94	47	0
`urlcount`	702	87/13/0/0	1.02	41	0
`usertrack`	409	81/19/0/0	1.00	6	0
`WebStone`	n/a	n/a	1.04		

Fig. 17. Apache module performance.

Name	Lines of Code	% sf/sq/w/rt	CCured Ratio	Valgrind Ratio	Memory Ratio	Lines Changed	Trusted Casts
pcnet32	1661	92/8/0/0	0.99			66	0
ping			1.00				
sbull	1013	85/15/0/0	1.00			18	0
seeks			1.03				
ftpd	6553	79/12/9/0	1.01	9.42	1.32	28	0
OpenSSL	177,426	67/27/0/6	1.40	42.9	1.81	2000	2
cast			1.87	48.7	3.56		
bn			1.01	72.0	3.47		
OpenSSH	65,250	70/28/0/3				365	14
client			1.22	22.1	3.88		
server			1.15		4.53		
sendmail	105,432	65/34/0/1	1.46	122	2.32	904	4
bind	336,660	79/21/0/0	1.81	129	3.84	224	237
tasks			1.11	81.4	1.86		
sockaddr			1.50	110	1.00		

Fig. 18. System software performance. A ratio of 1.03 means the CCured version was 3% slower than the original. Not all tests were applicable to Valgrind.

compiled and run using Linux 2.4.5. We used wrapper functions for Linux assembly code macros, which has the advantage of allowing us to insert appropriate run-time checks into otherwise opaque assembly (e.g., we performed bounds checks for the Linux internal `memcpy` routines). Some Linux macros (like `INIT_REQUEST`) and low-level casts were assumed to be part of the trusted interface. The performance measurements are shown in Figure 18. `pcnet32` measures maximal throughput, and "ping" indicates latency. `sbull` measures blocked reads (writes and character I/O were similar), and "seeks" indicates the time to complete a set number of random seeks.

Finally, we ran `ftpd-BSD` 0.3.2-5 through CCured. This version of `ftpd` has a known vulnerability (a buffer overflow) in the `replydirname` function, and we verified that CCured prevents this error. The biggest hurdle was writing a 70-line wrapper for the `glob` function. As Figure 18 shows, we could not measure any significant performance difference between the CCured version and the original. With both `ftpd` and Apache modules, the client and server were run on the same machine to avoid I/O latency.

8.2 Run-Time Type Information

In one of the first uses of the new RTTI pointer kind, we revisited one of our early experiments. With the original version of CCured, the ijpeg test in Spec95 had a slowdown of 115% due to about 60% of the pointers being WILD. (We also had to write a fair number of wrappers to address the compatibility problems.) This benchmark is written in an object-oriented style with a subtyping hierarchy of about 40 types and 100 downcasts. With RTTI pointers we eliminated all bad casts and WILD pointers with only 1% of the pointers becoming RTTI instead. This result shows how far the WILD qualifier can spread from bad casts. As shown in Figure 15, RTTI pointers reduce the slowdown to 50%.

We modified OpenSSL 0.9.6e, a cryptographic library and implementation of the Secure Sockets Layer protocol, to compile under CCured. Because of the structure of OpenSSL, this task required changing many function signatures so that they matched the types of the function pointers to which they were assigned. We used RTTI pointers extensively to handle OpenSSL's many uses of polymorphic pointers and container types. Because OpenSSL uses char* as the type for its polymorphic pointers, we were also forced to change the type of each of these pointers to void* to avoid unsound casts.[3] These changes allowed us to compile OpenSSL with only two "trusted" casts, which were needed for pseudo-random number seeds; thus, CCured should guarantee memory safety for this program with a minute trusted computing base. While running OpenSSL's test suite after compiling with CCured, we found one array bounds violation in the processing of rule strings. We also found a bounds error in the test suite itself and two programming errors in the library that do not affect program behavior.

Figure 18 shows the performance of OpenSSL's test suite after compiling with CCured, compared to the original C code. We show specific results for a test of the "cast" cipher and the big number package ("bn"). Note that the baseline C version was itself 20% slower than a default installation of OpenSSL, which uses assembly code implementations of key routines. CCured, of course, cannot analyze assembly code.

We also ran CCured on OpenSSH 3.5p1, an ssh client and server that links with the OpenSSL library. Not counting that library, we made 109 small changes and annotations to the 65,000 lines of code in OpenSSH. We used several trusted casts to deal with casts between different types of sockaddr structs, since CCured also adds bounds information to guarantee that these are used safely. We use an instrumented version of the OpenSSH daemon in our group's login server with no noticeable difference in performance. In doing so we have found one bug in the daemon's use of open().

We used CCured to make a type-safe version of sendmail 8.12.1. CCured is capable of preventing security-related errors in sendmail, including two separate buffer overrun vulnerabilities that have been found recently [CERT Coordination Center 2003]. Using CCured with sendmail required annotating variable-argument functions and replacing inline assembly with equivalent C code. To avoid WILD pointers, we modified several places in the code that

[3] With the adoption of ANSI C, void* replaces char* as the standard notation for an undetermined pointer type.

were not type safe: unions became structs, and unsound casts needed for a custom allocator were marked as trusted. We also used RTTI for polymorphic pointers that were used with dynamic dispatch. Finally, several stack allocated buffers were moved to the heap. In all, about 200 changes were required for the approximately 105,000 lines of code in sendmail. We found two bugs, both at compile time: a debug printf was missing an argument, and a (currently unused) section of code had a memory error due to a missing dereference operator. Figure 18 shows the results of a performance test in which messages were sent to a queue on the same host, using instrumented versions of sendmail for both client and daemon.

Finally, we ran CCured on bind 9.2.2rc1, a 330,000-line network daemon that answers DNS requests. CCured's qualifier inference classifies 30% of the pointers in bind's unmodified source as WILD as a result of 387 bad casts that could not be statically verified. (bind has a total of 82000 casts of which 26500 are upcasts handled by physical subtyping.) Once we turned on the use of RTTI, 150 of the bad casts (28%) proved to be downcasts that could be checked at run time. We instructed CCured to trust the remaining 237 bad casts rather than use WILD pointers, thereby trading some safety for the ability to use the more efficient SAFE and SEQ pointers. A security code review of bind should start with these 237 casts.

Figure 18 provides performance results for experiments involving name resolution; the "tasks" trial measured multiple workers, and the "sockaddr" trial measured IPv4 socket tasks. bind was the one of the most CPU-intensive pieces of systems software we instrumented, and its overhead ranged from 10% to 80%.

8.3 Compatible Pointer Representation

When curing bind, it was necessary to deal with networking functions that pass nested pointers to the C library, such as sendmsg and recvmsg. To demonstrate the benefit of our compatible pointer representation, we instructed CCured to use split types when calling such functions. By doing so, we eliminated the need to perform deep copies on the associated data structures, and we relieved the programmer of the burden of writing complex wrapper functions. The inference algorithm described in Section 6.2 determined that 6% of the pointers in the program should have split types and that 31% of these pointers needed a metadata pointer. The large number of metadata pointers was a result of the trusted casts used when curing bind; in order to preserve soundness when using these casts, we had to add metadata pointers to places where they would not normally be necessary.

We also used our compatible pointer representation when curing OpenSSH. As with bind, split types were used when calling the sendmsg function. In addition, we used split types when reading the environ variable, which holds the program's current environment. Less than 1% of all pointers in the program required a split type or a metadata pointer. The nature of the call sites allowed us to take advantage of split types without spreading them to the rest of the program.

To demonstrate the usefulness of our compatible pointer representation when linking with libraries that have complicated interfaces, we applied

CCured to the ssh client program *without* curing the underlying OpenSSL library. The ssh program uses 56 functions from the OpenSSL library, and many of these functions have parameters or results that contain pointers to pointers (and even pointers to functions). It would have been difficult to write wrappers for such a complex interface, but our compatible representation required the user to add only a handful of annotations (e.g., the user must identify places where results are returned via a function parameter). Even when using split types for all of these interfaces, our compatible representation was only needed in a limited number of places in the cured program: only 3% of pointers had split types, and only 5% of pointers required metadata pointers.

To determine the overhead of our compatible representation, we ran the olden, ptrdist, and ijpeg tests with all types split. In most cases, the overhead was negligible (less than 3% slowdown); however, execution times increased in a few cases. The em3d program (part of olden) was slowed down by 58%, and the anagram program (part of ptrdist) was slowed down by 7%. While split types are relatively lightweight, these outliers suggest that it is important to minimize the number of split types used, which can be achieved by applying our inference algorithm. Unfortunately, the performance impact of our compatible representation is difficult to predict at compile time; the slowdown appears to depend heavily on how the program uses pointers at run time. Finally, note that we could not measure the overhead of split metadata for programs such as bind and OpenSSH, since the split representation was necessary to cure and link these applications without writing a large number of complex wrapper functions.

8.4 Summary of Experiments

We have used CCured on several large, widely used programs for which reliability and security are critically important, including ftpd, bind, sendmail, OpenSSL, OpenSSH, and several Apache modules. Modifications and annotations were required to deal with unsound behavior in these programs. The performance of the instrumented code was far better than the performance when using existing tools such as Valgrind or Purify for adding memory safety to C. As a result, it is possible to use instrumented programs in day-to-day operations so that memory errors can be detected early and so that security holes can be prevented. Finally, we have detected several bugs in the programs we tested.

9. RELATED WORK

9.1 Safe C Language Designs and Extensions

There has been much interest in designing C-like languages or language subsets that are provably type safe. Smith and Volpano [1998] presented a polymorphic and provably type-safe dialect of C that includes most of C's features (including higher-order functions) but lacks casts and structures. Evans [1996] described a system in which programmer-inserted annotations and static checking techniques can find errors and anomalies in large programs. However, these approaches work only for programs written in the given dialect. CCured explicitly aims to bring safety to legacy applications.

The Cyclone language [Jim et al. 2002] is expressive, gives programmers a high degree of control, and has been used on similar types of programs (e.g., device drivers). Cyclone provides several features similar to CCured's, such as fat pointers and structural subtyping, along with many other features designed for new programs that are written in Cyclone. Unlike CCured, which uses pointer kind inference, a garbage collector, and other techniques to reduce the effort needed to compile legacy code, Cyclone's priority is to give programmers as much control as possible over performance.

9.2 Adding Run-Time Checks to C

There have been many attempts to bring some measure of safety to C by trading space and speed for security. Previous techniques have been concerned with spatial access errors (array bounds checks and pointer arithmetic) and temporal access errors (touching memory that has been freed), but none of them use a static analysis of the form presented here. Kaufer et al. [1988] presented an interpretive scheme called *Saber-C* that can detect a rich class of errors (including uninitialized reads and dynamic type mismatches but not all temporal access errors), but it runs about 200 times slower than normal. Austin et al. [1994] stored extra information with each pointer and achieved safety at the cost of a large (up to 540% speed and 100% space) overhead and a lack of backward compatibility. For example, it would fail to detect the multidimensional array bugs we found in go. Jones and Kelly [1997] stored extra information for run-time checks in a splay tree, allowing safe code to work with unsafe libraries; this scheme results in a slowdown factor of 5 to 6. The approaches of Austin et al. and Jones and Kelly are comparable to the implementation of CCured's `WILD` pointers. Patil and Fischer [1995] presented a system that uses a second processor to perform the bounds checks. The total execution overhead of a program is typically only 5% using their technique, but it requires a dedicated second processor. Since some of our benchmarks (e.g., apache) are multithreaded and could actually make use of a second processor, our overhead is actually lower in such cases. Loginov et al. [2001] stored type information with each memory location, incurring a slowdown factor of 5 to 158. This extra information allowed them to perform more detailed checks, and they could detect when stored types do not match declared types or when union members are accessed out of order. While their tool and ours are similar in many respects, their goal was to provide rich debugging information, whereas our goal is to make C programs safe while retaining efficiency.

Steffen's [1992] `rtcc` compiler is portable and adds object attributes to pointers. However, it fails to detect temporal access errors and does not perform any check optimizations. In fact, beyond array bounds check elimination, none of these techniques use type-based static analysis to aggressively reduce the overhead of the instrumented code.

9.3 Removing Dynamic Checks

Much work has been done to remove dynamic checks and tagging operations from Lisp-like languages. Henglein [1992] detailed a type inference scheme for removing tagging and untagging operations in Lisp-like languages. The overall structure of his algorithm is very similar to ours (simple syntax-direct constraint generation, constraint normalization, and constraint solving), but the domain of discourse is quite different because his base language is dynamically typed. In Henglein's system, each primitive type constructor is associated with exactly one tag, so there is no need to deal with the pointer/array ambiguity that motivates our `SEQ` pointers. In C, it is sometimes necessary to allocate an object of one type and later view it as having another type; Henglein's system disallows this behavior because tags are set at object creation time (that is, true C-style casts and unions are not fully supported [Jagannathan and Wright 1995]). Henglein was also able to sidestep update and aliasing issues because tagging and untagging create a new copy of the object (to which `set!` can be applied, for example); thus, programs never have tagged and untagged aliases for the same item. His algorithm does not consider polymorphism or module compilation [Kind and Friedrich 1993]. On the other hand, formal optimality results can be made [Henglein and Jorgensen 1994]. The CCured system uses a form of physical subtyping for pointers to structures, and it is not clear how to extend Henglein's constraint normalization procedure in such a case.

Jagannathan and Wright [1995] used a more expensive and more precise flow-sensitive analysis called *polymorphic splitting* to eliminate run-time checks from higher-order call-by-value programs. Shields et al. [1998] presented a system in which dynamic typing and staged computation (run-time code generation) coexist: all deferred computations have the same dynamic type at compile-time and can be checked precisely at run-time. Such a technique can handle persisting dynamic data, a weakness of our current system. Soft type systems [Cartwright and Fagan 1991] also infer types for procedures and data structures in dynamically typed programs. Advanced soft type systems [Wright and Cartwright 1997] can be based on inclusion subtyping and can handle unions, recursive types, and other complex language features. Finally, Kind and Friedrich [1993] presented a practical ML-style type inference system for Lisp. As with Henglein [1992], such systems start with a dynamically typed language and thus tackle a different core problem.

Our constraint generation process effectively computes an alias set similar to that of Steensgaard [1996]. The `EQ` constaints are similar to his unification rules. We also track additional information (e.g., pushing `SEQ` qualifiers back along assignments via `BOUNDS` constaints) that is important for our analysis. The analysis is thus conceptually similar to that of Das [2000] in that assignments are treated directionally while pointers "one level down" are unified. When viewed solely as an alias analysis, however, our results are much less precise than those of Das.

9.4 Dynamic Values

An entire body of research [Cartwright and Fagan 1991; Henglein 1992; Kind and Friedrich 1993; Shields et al. 1998; Thatte 1990; Wright and Cartwright 1997] has examined the notion of a Dynamic type whose values are ⟨type, ptr⟩ packages. Such a value can only be used by first extracting and checking the type. In particular, one can only write values that are consistent with the packaged type. Because the underlying value's static type is carried within the Dynamic package and checked at every use, there is no problem with Dynamic aliases for statically typed data.

This approach is in contrast to CCured's `WILD` pointers, which allow values of arbitrary type to be written and which allow arbitrary interpretation of the value read (except that the tags prevent misinterpreting a pointer base field). Thus, a memory word's type may change during execution. This flexibility is built into CCured because we expect some C programs to allocate large areas of memory and reuse that memory in different ways. However, its cost is that `WILD` must be a closed world, with no aliases of statically typed data.

The inference algorithm in CCured bears some resemblance to Henglein's [1992] inference algorithm, but we consider physical subtyping, pointer arithmetic, and updates. Henglein's algorithm has the nice feature that it does not require any type information to be present in the program. We believe that his algorithm does not extend to the more complex language we consider here and that existing C types contain valuable information that should be used to make inference both simpler and predictable (in terms of when a pointer will be inferred `WILD`).

Abadi et al. [1991] studied the theoretical aspects of adding a Dynamic type to the simply typed λ-calculus, and they discussed extensions to polymorphism and to abstract data types. CCured's `RTTI` qualifier is similar, but we combine it with an inference algorithm based on physical subtyping.

Thatte [1990] extended the work of Abadi et al. [1991] to replace the `typecase` expressions with implicit casts. Their system does not handle reference types or memory updates, and `Dynamic` types are introduced to add flexibility to the language. In contrast, our system handles memory reads and writes, allows `WILD` values to be manipulated (e.g., via pointer arithmetic) without checking their tags, and uses `WILD` types to guarantee the safety of code that cannot be statically verified.

The programming languages CLU [Liskov et al. 1981], Cedar/Mesa [Lampson 1983], and Modula-{2+,3} [Cardelli et al. 1989] include similar notions of a dynamic type and a typecase statement. This idea can also be seen in CAML's exception type [Remy and Vouillon 1997].

9.5 Physical Subtyping

Another line of research tries to find subsets of C that can be verified as type-safe at compile time. Ramalingam et al. [1999] have presented an algorithm for finding the coarsest acceptable type for structures in C programs. Chandra and Reps [1999] presented a method for physical type checking of C programs based on structure layout in the presence of casts. Their inference method can reason about casts between various structure types by considering the physical layout of memory. Many real-world examples fail to type check in their system for the same reason that we must mark some pointers `WILD`: their safety cannot be guaranteed at compile time. Siff et al. [1999] reported that many casts in C programs are safe upcasts, and they presented a tool to check such casts. Each of these approaches requires programs to adhere to their particular subset; otherwise, the program is rejected. CCured's static type system has comparable expressiveness, but CCured can fall back on its flexible RTTI or WILD pointers to handle the corner cases. Our notion of physical subtyping extends this line of work to include pointer arithmetic (see Section 3.1). Most such type systems and inference methods are presented as sources of information. In this article, we present a type system and an inference system with the goal of making programs safe.

9.6 Compatibility

The global splay tree used by Jones and Kelly [1997] provides an alternative approach to the problem of library compatibility; however, we found that looking up metadata in a global data structure was prohibitively expensive. Also, Patil and Fischer [1995] maintained shadow data using a technique that resembles our compatible metadata representation. However, CCured's representation handles different kinds of metadata for different pointer kinds, requires less overhead, and allows run-time checking to be done in the same processor and address space as the main program. Furthermore, in CCured it is possible for both the compatible representation and the more efficient incompatible representation to coexist in a given program.

A number of authors have studied intensional polymorphism [Harper and Morrisett 1995; Crary et al. 1998; Duggan 1999], which is an approach to compiling polymorphism that allows type information to be used at run-time. These techniques can allow a compiler to use efficient data representations while preserving type safety. CCured has two possible data representations, but instead of generating polymorphic code that handles both representations, we require SPLIT types to be used whenever data may potentially be passed to an external library. This approach is reasonable in our case, since our inference algorithm is effective in limiting the spread of SPLIT qualifiers. Also, many of these approaches to intensional polymorphism represent types as terms in parallel with expressions, which resembles our split representation. However, CCured's split representation is used to carry array bounds for the purpose of validating pointers, rather than to carry type information for the purpose of implementing a form of polymorphism.

10. CONCLUSIONS

CCured is a C program analysis and transformation system that ensures memory safety. It first analyzes the program and attempts to find safe portions of it that adhere to a strong type system. The remainder of the program is instrumented with run-time checks. Parts of the program that cannot be proved safe statically are often slow and incompatible with external libraries. The techniques in this article improve the usability of CCured by increasing the amount of the program that can be verified statically and the ease with which instrumented code can interface with the outside world.

Physical subtyping prevents many type casts from requiring the use of WILD pointers. We have incorporated physical subtyping with pointer arithmetic, allowing upcasts (which make up about 33% of all casts) to be statically verified as safe. This approach improves the analysis portion of CCured.

We have described a system for run-time type information that handles downcasts, and we provide an inference algorithm that uses physical subtyping to decide which pointers require this information. As a result, CCured can reason about the common idioms of parametric and subtype polymorphism. Using this mechanism improves the analysis portion of CCured and adds additional run-time checks. When run-time type information is combined with physical subtyping, more than 99% of all program casts can be verified without resorting to WILD pointers.

CCured's pointers are often incompatible with external libraries. One way to bridge this gap is by writing wrappers, and we have extended CCured to include support for writing wrappers that ensure memory safety. In addition, we presented a scheme for splitting CCured's metadata into separate data structures, allowing instrumented programs to invoke external functions directly. This mechanism could also be useful for any run-time instrumentation scheme that must maintain metadata with pointers while maintaining compatibility with precompiled libraries.

We verified the utility of these extensions while working on a number of real-world security-critical network daemons, device drivers, and Web server modules. Without these extensions, these programs would have been quite difficult to make safe using CCured. Equipped with the mechanisms described in this article, we can build tools, such as CCured, that are better able to analyze and instrument real-world software systems, thereby improving their reliability and security.

ACKNOWLEDGMENTS

We thank Aman Bhargava, SP Rahul, and Raymond To for their contributions to the CIL infrastructure. We also thank Alex Aiken, Ras Bodik, Jeff Foster, the members of the Open Source Quality group, and the anonymous referees for their advice and helpful comments on this article.

REFERENCES

ABADI, M., CARDELLI, L., PIERCE, B., AND PLOTKIN, G. 1991. Dynamic typing in a statically typed language. *ACM Trans. Prog. Lang. Syst. 13*, 2 (April), 237–268.

AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. 1994. Efficient detection of all pointer and array access errors. *SIGPLAN Not. 29*, 6 (June), 290–301. Also in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

BOEHM, H.-J. AND WEISER, M. 1988. Garbage collection in an uncooperative environment. *Softw.—Pract. Exper. 18*, 9, 807–820.

CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G. 1989. Modula-3 report (rev.). SRC Research rep. 52. Digital Equipment Corporation Systems Research Center, Palo alto, CA.

CARLISLE, M. C. 1996. Olden: Parallelizing programs with dynamic data structures on distributed-memory machines. Ph.D. dissertation. Princeton University Department of Computer Science, Princeton, NJ.

CARTWRIGHT, R. AND FAGAN, M. 1991. Soft typing. In *Proceedings of the '91 Conference on Programming Language Design and Implementation*. 278–292.

CERT Coordination Center. 2003. CERT Advisory CA-2003-12: Buffer overflow in sendmail. Web site: `http://www.cert.org/advisories/CA-2003-12.html`.

Chandra, S. and Reps, T. 1999. Physical type checking for C. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Software Engeneering Notes (SEN), vol. 24.5. ACM Press, New York, NY, 66–75.

Condit, J., Harren, M., Necula, G. C., McPeak, S., and Weimer, W. 2003. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. ACM, Press, New York, NY, 232–244.

Crary, K., Weirich, S., and Morrisett, J. G. 1998. Intensional polymorphism in type-erasure semantics. In *Proceedings of the International Conference on Functional Programming*. 301–312.

Das, M. 2000. Unification-based pointer analysis with directional assignments. In *Proceedings of the Conference on Programming Language Design and Implementation*.

Duggan, D. 1999. Dynamic typing for distributed programming in polymorphic languages. *ACM Trans. Prog. Lang. Syst. 21*, 1, 11–45.

Evans, D. 1996. Static detection of dynamic memory errors. *ACM SIGPLAN Not. 31*, 5, 44–53.

Harper, R. and Morrisett, G. 1995. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, CA). 130–141.

Hastings, R. and Joyce, B. 1991. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Usenix Winter 1992 Technical Conference*. Usenix Association, Berkeley, CA, 125–138.

Henglein, F. 1992. Global tagging optimization by type inference. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. 205–215.

Henglein, F. and Jorgensen, J. 1994. Formally optimal boxing. In *Proceedings of the 21th Annual ACM Symposium on Principles of Programming Languages*. ACM, 213–226.

Hirzel, M. 2000. Effectiveness of garbage collection and explicit deallocation. M.S. thesis. University of Colorado at Boulder, Boulder, CO.

ISO/IEC. 1999. ISO/IEC 9899:1999(E) Programming Languages—C. ISO/IEC, Geneva, Switzerland. Web site: `www.iso.ch`.

Jagannathan, S. and Wright, A. 1995. Effective flow analysis for avoiding run-time checks. In *Proceedings of the Second International Static Analysis Symposium*, Vol. 983. Springer-Verlag, Berlin, Germany, 207–224.

Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y. 2002. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference* (Monetery, CA).

Jones, R. W. M. and Kelly, P. H. J. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automatic Debugging* (May). 13–26.

Kaufer, S., Lopez, R., and Pratap, S. 1988. Saber-C: An interpreter-based programming environment for the C language. In *Proceedings of the Summer Usenix Conference*. 161–171.

Kind, A. and Friedrich, H. 1993. A practical approach to type inference for EuLisp. *Lisp Symbol. Computa. 6*, 1/2, 159–176.

Lampson, B. 1983. A description of the Cedar language. Tech. rep. CSL-83-15. Xerox Palo Alto Research Center, Palo Alto, CA.

Liskov, B., Atkinson, R. R., Bloom, T., Moss, E. B., Schaffert, R., and Snyder, A. 1981. *CLU Reference Manual*. Springer-Verlag, Berlin, Germany.

Loginov, A., Yong, S., Horwitz, S., and Reps, T. 2001. Debugging via run-time type checking. In *Proceedings of FASE 2001: Fundamental Approaches to Software Engineering*.

Necula, G. C., McPeak, S., and Weimer, W. 2002a. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*. ACM, Press, New York, NY, 128–139.

Necula, G. C., McPeak, S., and Weimer, W. 2002b. CIL: Intermediate language and tools for the analysis of C programs. In *Proceedings of the International Conference on Compiler Construction* (Grenoble, France). 213–228. Available online from `http://raw.cs.berkeley.edu/Papers/`.

Patil, H. and Fischer, C. N. 1995. Efficient run-time monitoring using shadow processing. In *Proceedings of the Conference on Automated and Algorithmic Debugging*. 119–132.

PATIL, H. AND FISCHER, C. N. 1997. Low-cost, concurrent checking of pointer and array accesses in C programs. *Softw.—Pract. Exper. 27*, 1 (Jan.), 87–110.

RAMALINGAM, G., FIELD, J., AND TIP, F. 1999. Aggregate structure identification and its application to program analysis. In *Proceedings of the Symposium on Principles of Programming Languages*. 119–132.

REMY, D. AND VOUILLON, J. 1997. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the Symposium on Principles of Programming Languages*. 40–53.

SECURITEAM.COM. 2000. PHP3/PHP4 format string vulnerability. Web site: `http://www.securiteam.com/securitynews/6O00T0K03O.html`.

SEWARD, J. 2003. Valgrind, an open-source memory debugger for x86-GNU/Linux. Tech. rep. Available online at `http://developer.kde.org/˜sewardj/`.

SHIELDS, M., SHEARD, T., AND JONES, S. L. P. 1998. Dynamic typing as staged type inference. In *Proceedings of the Symposium on Principles of Programming Languages*. 289–302.

SIFF, M., CHANDRA, S., BALL, T., KUNCHITHAPADAM, K., AND REPS, T. 1999. Coping with type casts in C. In *1999 ACM Foundations on Software Engineering Conference*. Lecture Notes in Computer Science, vol. 1687. Springer-Verlag, Berlin, Germany, ACM Press, New York, NY, 180–198.

SMITH, G. AND VOLPANO, D. 1998. A sound polymorphic type system for a dialect of C. *Sci. Comput. Prog. 32*, 1–3, 49–72.

SPEC. 1995. Standard Performance Evaluation Corporation Benchmarks. Web site: `http://www.spec.org/osg/cpu95/CINT95`.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the Symposium on Principles of Programming Languages*. 32–41.

STEFFEN, J. L. 1992. Adding run-time checking to the Portable C Compiler. *Softw.—Pract. Exper. 22*, 4 (Apr.), 305–316.

THATTE, S. 1990. Quasi-static typing. In *Proceedings of the Conference record of the 17th ACM Symposium on Principles of Programming Languages (POPL)*. 367–381.

WAGNER, D., FOSTER, J., BREWER, E., AND AIKEN, A. 2000. A first step toward automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network Distributed Systems Security Symposium*. 1–15.

WRIGHT, A. AND CARTWRIGHT, R. 1997. A practical soft type system for Scheme. *ACM Trans. Prog. Lang. Syst. 19*, 1 (Jan.), 87–152.

Received April 2003; revised November 2003; accepted May 2004